Adaptive Consistency Management for In-memory Storage

Inaugural-Dissertation

zur Erlangung des Doktorgrades
der Mathematisch-Naturwissenschaftlichen Fakultät
der Heinrich-Heine-Universität Düsseldorf

vorgelegt von

Kim-Thomas Rehmann geb. Möller
aus Hameln

Düsseldorf, Mai 2013

Aus dem Institut für Informatik
der Heinrich-Heine-Universität Düsseldorf

Gedruckt mit der Genehmigung der
Mathematisch-Naturwissenschaftlichen Fakultät der
Heinrich-Heine-Universität Düsseldorf

Referent: Prof. Dr. Michael Schöttner
Korreferent: Prof. Dr. Martin Mauve
Korreferent: Prof. Dr. Franz J. Hauck

Tag der mündlichen Prüfung: 8. Mai 2013

Abstract

The availability of storage media with high capacity at low prices has recently increased the demand for software applications that are able to analyze large data volumes. Engineers build large-scale storage systems using both scale-up and scale-out techniques. Scale-up increases the amount of data a single nodes stores, whereas scale-out aggregates the capacity of several servers and increases the peak throughput of data transfers. Scale-out systems do not share any resources except for a communication bus, such that the participating compute nodes need to share information explicitly. High-speed communication in local-area networks and in-memory storage of information reduce the accesses latency compared to storage on harddisks.

Traditional application designs are often unable to use large-scale storage efficiently. Parallelization of sequential programs faces problems of data interdependencies, distribution unawareness and error-proneness in distributed settings. Design patterns that are successful for sequential applications often do not apply any more for concurrent execution. The success of a storage service depends largely on the acceptance by application developers. Thus, such a service must adhere to convenient design thinking while at the same time it should not degrade the performance of optimized applications.

This thesis presents novel approaches to accommodate application programming through appropriate design of the storage service. By combining techniques from storage replication, peer-to-peer computing and optimistic synchronization, the proposed storage service relieves application programmers from handling failures and explicit lock management. Optimizations that are mostly transparent for the application allow to reduce false sharing effects and to increase storage utilization. Specifically, the thesis details the design and implementation of two adaptive techniques to improve the performance of distributed transactional memory. Adaptive replication makes storage objects available rapidly and increases update throughput by analyzing object access patterns. Adaptive conflict granularity allows bulk object transfers while at the same time detecting and avoiding false sharing situations. The described techniques simplify application programming by improving the context-awareness of distributed storage services. This thesis also introduces a framework for in-memory applications that adhere to the MapReduce programming model.

The use and applicability of the suggested enhancements for a scalable storage service are exemplified with a number of applications from diverse problem domains including computer graphics, statistics and data mining. The examples also serve to analyze the performance and scalability of the storage service. The measurements demonstrate that the extensions improve the access parallelism of in-memory storage without complicating the programming model or increasing storage requirements.

In summary, this thesis presents several contributions to the research field of large-scale in-memory data management. The evaluation of the contributions proves their applicability and potential for realistic workload.

Zusammenfassung

Die Verfügbarkeit von Speichermedien hoher Kapazität zu niedrigen Kosten hat in den letzten Jahren die Nachfrage nach Softwareanwendungen zur Verarbeitung großer Datenmengen gesteigert. Ingenieure setzen Scale-Up- und Scale-Out-Techniken ein, um große Speichersysteme zu entwickeln. Scale-Up erhöht die Speicherkapazität pro Rechner, wohingegen Scale-Out mehrere Rechner zusammenfasst, um ein größeres Speichervermögen zu erreichen und den Datenduurchsatz zu steigern. Scale-Out-Speichersysteme besitzen nur einen Speicherbus als gemeinsame Komponente, so dass die teilnehmenden Rechenknoten Informationen explizit verteilen müssen. Schnelle Kommunikation in lokalen Netzwerken und die In-Memory-Speicherung von Informationen verringern die Zugriffslatenz verglichen mit der Speicherung auf Festplatten.

Herkömmliche Entwurfsmuster für Softwareanwendungen sind häufig nicht in der Lage, große Speichersysteme effizient zu nutzen. Zum Beispiel wird die Parallelisierung sequentieller Programme durch Datenabhängigkeiten, mangelnde Kenntnis der Datenverteilung und fehleranfällige verteilte Szenarien erschwert. Entwurfsmuster, die erfolgreich für sequentielle Programme eingesetzt werden, lassen sich auf nebenläufige Ausführung häufig nicht anwenden. Der Markterfolg eines Speicherdienstes hängt zum Großteil von der Akzeptanz durch die Anwendungsentwickler ab. Ein nützlicher Speicherdienst muss einerseits einem praktischen Entwurfsdenken entgegenkommen, darf aber andererseits nicht die Leistung optimierter Anwendungen beeinträchtigen.

Diese Arbeit stellt neuartige Ansätze vor, um die Programmierung von Anwendungen durch einen geeignet entworfenen Speicherdienst zu unterstützen. Indem der vorgeschlagene Datenspeicher Techniken des Peer-To-Peer-Computing und der optimistischen Synchronisierung kombiniert, erleichtert er den Anwendungsprogrammierern das Behandeln von Fehlern und vermeidet explizite Sperrverfahren. Optimierungen, die größtenteils transparent für Anwendungen sind, ermöglichen es, False-Sharing-Effekte zu reduzieren und den zur Verfügung stehenden Speicher gut auszunutzen. Insbesondere beschreibt diese Arbeit Entwurf und Implementierung zweier adaptiver Techniken, die die Leistungsfähigkeit eines verteilten transaktionalen Speichers erhöhen. Die adaptive Replikation von Speicherobjekten reduziert die Zugriffslatenz und erhöht den Aktualisierungsdurchsatz durch die Analyse von Objektzugriffsmustern. Adaptive Konfliktgranularität ermöglicht Massentransfers, wobei False-Sharing-Effekte erkannt und vermieden werden. Die beschriebenen Techniken vereinfachen die Anwendungsprogrammierung, indem die Kontextabhängigkeit des verteilten Speicherdienstes verbessert wird. Diese Arbeit präsentiert auch ein Programmiergerüst für In-Memory Anwendungen, die dem MapReduce-Programmiermodell entsprechen.

Eine Anzahl von Anwendungen aus verschiedenen Einsatzgebieten von Computergrafik über Statistik bis hin zu Data Mining illustriert, dass die vorgeschlagenen Verbesserungen anwendbar und wirksam sind. Diese Beispielanwendungen dienen auch dazu, den Prototypen eines verbesserten Speicherdienstes in Bezug auf Skalierbarkeit und Leistungsfähigkeit zu evaluieren. Anhand von Messungen wird gezeigt, dass die Verbesserungen die Skalierbarkeit des In-memory Speichers erhöhen, ohne das Programmiermodell zu verkomplizieren oder den Speicherbedarf zu erhöhen.

Insgesamt enthält diese Arbeit mehrere Beiträge zum Forschungsgebiet der skalierbaren In-Memory-Datenverwaltung. Die Auswertung der Beiträge belegt ihre Anwendbarkeit und die Möglichkeiten für realistische Nutzlast.

Danksagung

Der Betreuer meines Promotionsvorhabens, Herrn Prof. Dr. Michael Schöttner, hat mir die Möglichkeit gegeben, verteilte Systeme zu erforschen, die Lehre in der Abteilung Betriebssysteme mitzugestalten und dabei die vorliegende Dissertation anzufertigen. Für das mir entgegengebrachte Vertrauen, die fachliche und persönliche Unterstützung bedanke ich mich herzlich. Mein Dank gilt auch den Herren Prof. Dr. Martin Mauve und Prof. Dr. Franz J. Hauck für ihr Interesse und ihre Zeit, diese Arbeit zu begutachten.

Meinen Bildungsweg konnte ich nur dank der Unterstützung meiner Eltern verfolgen. Dafür bin ich ihnen überaus dankbar. Weiterhin danke ich meinen Brüdern Martin und Kai-Christian, meiner Tante Roswitha, meinen Schwiegereltern Heidemarie und Kurt und meiner Schwägerin Petra für ihre Unterstützung.

Gruß und Dank an meine ehemaligen Kollegen Marc-Florian, John, Michael S., Florian und Michael B. für die gute Zusammenarbeit und ganz besonders Angela für ihren organisatorischen Beistand. Den von mir mitbetreuten Studierenden Eugen, Patrick, Markus, Christoph, Moritz, Mario, Martin M., Dennis, Roman, Hoang, Dirk, Christian L., Serdar, Pierre, Prashanna, Daniel, Oliver, Martin T., Christian W. und Kevin danke ich für die vielfältigen Diskussionen und Implementierungen. Für die freundliche Aufnahme in die produktive und innovationsfreudige Arbeitsatmosphäre der TIP-HANA-Entwicklungsabteilung danke ich meinen Kollegen von der SAP AG.

Ohne meine Frau Sonja wäre diese Dissertation nicht entstanden. Neben ihrer eigenen Karriere findet sie immer die Kraft, mich zu unterstützen. Ich wünsche uns noch viele weitere erlebnisreiche, gemeinsame Jahre.

Contents

1

Introduction

The prominent paradigm in computer science today is *cloud computing*, that is, provisioning of information technology services such as data storage and application execution over the Internet. Cloud computing providers lend infrastructure, platforms, and software to companies and individuals, which use these services on demand, consuming and paying just what they actually need.

A multitude of technologies to store, process and transfer information have been developed in the recent years. Early ad-hoc solutions such as storage services, virtualized execution and web services are currently transforming into industry standards. However, researchers are still working eagerly to solve yet unaddressed problems resulting from new application areas. The fast pace of industrial development demonstrates that technical improvements are still achievable.

The price of storage technologies and the engineering effort for new cloud applications are important factors in the total cost of ownership. Therefore, more powerful and efficient data processing techniques have a high economic relevance, which becomes evident from the current popularity of the *big data* trend. This thesis focuses on data storage, a central aspect of cloud computing. The adaptive data-oriented communication techniques discussed are highly relevant for distributed applications and storage systems.

1.1 Motivation

The availability of storage media with high capacity at low prices has recently increased the demand for software applications that are able to analyze large data volumes. Engineers build large-scale storage systems using both scale-up and scale-out techniques [60]. Scale-up increases the amount of data a single nodes stores, whereas scale-out aggregates the capacity of several servers. Scale-out systems do not share any resources except for a communication bus, such that the participating compute nodes need to share information explicitly. In-memory storage of information reduces the accesses latency compared to storage on harddisks [76, 146].

Distributed applications are characterized by geographically dispersed users and by independent computing nodes at different locations. Traditional application designs are often unable to use distributed resources and nodes efficiently. Parallelization of sequential programs faces problems of data interdependencies, distribution unawareness and error-proneness in distributed settings. Moreover, limited data throughput and high communication latencies complicate the cooperation of participants in a distributed application. Therefore, cloud computing abstracts from individual resources and offers storage capacity as a service to distributed applications.

Distributed computing defines two opposed paradigms for information flow between nodes participating in a distributed application. With message-centric communication, data packets are sent from source nodes to destination nodes. Contrarily, data-centric communication places data objects in a distributed storage service, where nodes can store and retrieve information.

Message-centric communication allows explicit control over information flow. For example, in the peer-to-peer computing paradigm, all participants receive and forward messages on behalf of their peers in order to build a fully decentralized collaboration network. However, the precise control over communication comes at the cost of individual solutions for similar applications, higher engineering overhead and an increased rate of programming errors.

A data-centric communication service is neutral with respect to applications, which means that the same service can provide its communication facilities to different applications. The focus on data, for example inputs and results of calculations, matches the intuition of information processing. However, a severe problem of data-centric communication is that applications lacks direct control over how information spreads through the distributed system. For example, the data consistency that the storage guarantees towards the application is typically statically defined. Both paradigms are not exact antipodes, but they can be used in combination. With respect to these communication paradigms, this thesis focuses on data-centric communication and contributes several new approaches to improve storage performance without complicating application development.

Applications have various requirements on the storage service they use. Functional requirements specify what operations the service shall provide. Non-functional requirements describe the quality of service (QoS) expected by applications. QoS parameters include the latency of storage operations, how the storage should handle situations where hardware is failing or where the operations of several nodes interact. Fundamental notions to describe storage properties are *data consistency*, *availability* and *tolerance of network partitions*. The following definitions are aligned with the semantics described by Eric Brewer [36].

Definition 1 *Data consistency is a contract between a distributed application and the underlying storage that describes when the storage makes modifications of stored data visible to the participants of the application.*

The issue of data consistency arises from the fact that a distributed system lacks a global timesource having arbitrary high precision, such that the ordering of events is sometimes undefined. The strongest level of distributed data consistency equals the effect of all operations working on a central copy of the data. Strong consistency usually imposes a higher communication overhead than weak consistency.

Definition 2 *Availability describes the responsiveness of read and write operations.*

Availability is essentially a binary property in the sense that an operation either succeeds or fails eventually. It is not concerned with the timing behavior of these operations.

Definition 3 *Network partitioning denotes the existence of disconnected groups of nodes.*

Partitioned nodes cannot communicate with each other, neither directly nor via indirect connections over several hops. Partitions result from failures in hardware or software components that the storage service builds upon. They are commonly detected using timeout mechanisms. Network partitioning should not be confused with data partitioning, which places storage objects or parts of objects on different node.

Consistency, availability and partition tolerance are disparate properties according to Brewer's CAP (consistency, availability and partition tolerance) theorem [83], which has been proven by Gilbert and Lynch [89]. Figure 1.1 illustrates the diverging goals of the three concepts. As long as the network of nodes is connected, the storage service can guarantee consistency and availability. In case of a network partition, the storage has to decide whether it guarantees availability and therefore relaxes data consistency, or whether it enforces consistency and thus risks that operations on objects do not terminate. In addition to the CAP theorem, the figure also depicts the contrast between partition tolerance and latency, which Daniel Abadi has recently described [1]. If an application tolerates arbitrarily high latencies of operations, network partitions are only a theoretical problem. However, an upper bound on the latency of operations effectively limits the partition tolerance.

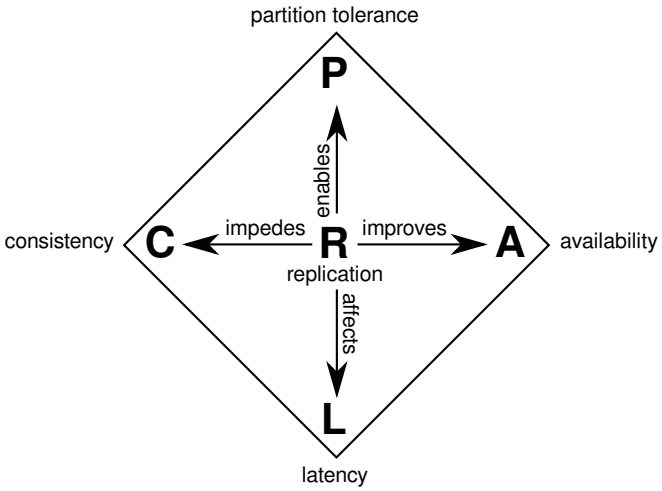


Figure 1.1: Diverging goals of consistency, availability and network partition tolerance

The centerpiece of the four concepts *consistency, availability, partition tolerance* and *latency* is data replication.

Definition 4 *Replication is the action of duplicating objects at different nodes in a way that the copies, which are called replicas, are indistinguishable from the original.*

On the one hand, replication can improve data availability and thereby help reduce latency and tolerate network partitions. On the other hand, replication impedes data consistency, such that it hinders partition tolerance and complicates achieving low latency in case of frequent data updates. Legacy database management systems (DBMS) define transaction semantics, which entails the *atomicity, consistency, isolation* and *durability* (ACID) properties. ACID allows for optimistic concurrency control [121], which means that conflicting storage operations can be resolved by restarting transactions. As an alternative to the strict ACID properties, Brewer suggests the properties *basically available, soft state* and *eventual consistency* (BASE). Compared to the ACID properties, BASE builds on data replication in order to reduce access latency under weakened consistency guarantees.

Considering the difficulties arising from the CAP theorem and the quest for low latency of operations, a single storage service that automatically fits all application requirements cannot exist. To be usable with a wide range of applications, a storage service must implement an interface that allows applications to specify their service level requirements. However, the configurability should not complicate application development.

Currently trending programming models for highly distributed and parallel computations such as MapReduce [59], Cilk [30] and Dryad [107] are good examples for the relation between storage service and application. On the one hand, programming models simplify the development of applications by allowing the reuse of common functionality such as job control and storage access. On the other hand, programming models limit the degrees of freedom to structure applications. With respect to the storage service, a programming model can enforce certain data access patterns, which the storage service can optimize, but the model can also get in the way of non-intended accesses. Thus, the quest for distributed storage services and programming models is to provide scalable services to a wide range of applications without requiring customization of applications, storage service or programming framework.

1.2 Target application domains

The huge storage and processing capabilities of today's distributed systems enable fast analytical processing of large data sets. This thesis projects a storage service to exemplify the proposed enhancements for adaptivity. Use cases from different problem domains evaluate the practical applicability of

the storage service. The applications include text, image and hypertext processing, descriptive statistics as well as optimization algorithms. To demonstrate how unmodified applications can benefit from the enhancements, the thesis also describes the design and implementation of a filesystem based on the exemplary storage service.

1.3 Contribution of this thesis

The field of distributed in-memory storage for parallel applications opens a wide variety of research topics, many of which have not been addressed yet. This thesis addresses the tension between diverse requirements and simplified programming of applications by suggesting flexible and adaptive approaches to distributed storage. By combining techniques from in-memory storage replication, peer-to-peer computing and optimistic concurrency control, the proposed storage service relieves application programmers from handling failures and explicit but error-prone lock management. Optimizations that are mostly transparent for the application allow to reduce false sharing effects and to increase storage utilization.

The allocation strategy of objects is fundamental to the scalability of a distributed storage system. Thus, this thesis proposes a novel, scalable strategy that takes into account dynamic sizing and load balancing in face of frequent failures (see Chapter 2). To increase the flexibility of a storage service, it contributes a hybrid mechanism for access control, which gives applications the choice between manipulation of objects using API functions and accessing dynamic objects directly as if they were allocated by malloc. that unifies memory-mapped and function-based access control mechanisms.

The recent development of storage systems for specific application areas suggests that static coherence and consistency protocols are not flexible enough. To improve the semantic sensitivity of generic distributed in-memory storage, this thesis suggests adaptive mechanisms for replication and for conflict unit granularity. Smart replication of storage objects decreases access latency and increases update throughput by analyzing object access patterns (see Chapter 4). Adaptive conflict granularity allows for bulk object transfers. At the same time, it detects and avoiding false sharing situations (see Chapter 5).

MapReduce is a popular model for data-intensive computations. This thesis suggests using in-memory storage for distributed MapReduce and demonstrates that in-memory storage not only simplifies the data access for applications, but also allows to conveniently implement job management with load balancing (see Chapter 6). Example applications for in-memory MapReduce include computer graphics, statistics and data mining. The examples also serve to analyze the performance and scalability of the storage service.

In-memory storage usually offers a procedural object-based interface, which provides calls to create, access, modify and destroy shared objects. An alternative to object-based procedures is the long-standing filesystem interface. Chapter 7 describes how to implement a distributed filesystem based on transactional in-memory storage. The user-level implementation allows the filesystem to integrate seamlessly into the operating system interface and to implement specific interface extensions for customized applications.

In order to demonstrate that the suggested storage components and procedures are effective, this thesis evaluates the prototype for a in-memory storage system in terms of measures such as latency, storage consumption and throughput. The measurements in Chapter 8 entail artificial workload as well as diverse example applications.

In summary, this thesis presents several contributions to the flexible and adaptive management of large-scale in-memory storage systems. The evaluation of the contributions proves their applicability and potential for realistic workload.

1.4 Outline

The data structures and metadata management that a distributed storage service builds upon impact scalability, performance and resilience. Chapter 2 introduces and defines central concepts such as stor-

age nodes and objects. The chapter also describes the stacking of allocators for dynamic management of small objects in a distributed system as well as the coexistence of different access control mechanisms.

As mentioned before, a storage system must find an appropriate compromise between consistency and performance. A state-of-the-art distributed transactional memory is described in Chapter 3. In addition, the chapter summarizes approaches to weaken the semantics of transactional consistency.

Replication is a key mechanism to achieve better performance in a distributed system. Chapter 4 introduces several concepts and mechanisms of storage replication. Then it describes the integration of a generic replication service with consistency models. Finally, it presents a strategy to switch dynamically and adaptively between update and invalidate coherence. Chapter 5 addresses the problem of conflict probability for concurrent accesses. It describes an algorithm to adaptively adjust cache granularity to counteract false sharing situations.

In addition to smart replication and caching granularity strategies, well-structured algorithm design can help improve scalability. Chapter 6 builds upon the popular MapReduce programming model and suggests an in-memory framework for regular as well as extended MapReduce workload. Filesystems provide an established interface for data exchange at the operating system level. Chapter 7 presents the design and implementation of an in-memory filesystem based on a distributed storage service.

Chapter 8 describes the execution and evaluation of the previously mentioned strategies and applications. Finally, Chapter 9 concludes this work with a summary and an outlook on future research directions.

2

Elastic management of distributed memory objects

Handling distributed state is an important aspect in the implementation of distributed applications. In order to simplify interaction with physical storage and avoid reimplementation of data sharing, applications benefit from accessing data through storage software that abstracts from hardware peculiarities and internal organization of the storage. A storage service provides a generic interface hiding responsibilities such as data transfer and location, fault handling and internal organization of data.

A typical large-scale storage service, such as a data store for use by cloud computing applications, is faced with several aspects of dynamic behavior. First, data durability requires that users and components of the storage must be able to join and leave during runtime. Some distributed storage services such as Cassandra were even designed under the assumption that failures are not the exception but the common case [122]. Second, the storage requirements of most applications are not constant but vary significantly over time. Applications allocate distributed objects to store more information, and they release unused objects in order to avoid storage shortage. Third, a relational database is sometimes inflexible in representing a data model that changes dynamically over time. Many cloud applications prefer a NoSQL data model, which allows to store arbitrarily structured information for a given key. Cloud computing uses the notion *elasticity* to describe a system that is able to adapt to changing workload.

This chapter considers design alternatives for an elastic storage service. Many definitions and explanations are relevant for the subsequent chapters. The chapter is structured as follows. First, it discusses approaches to create, identify and destroy dynamic memory objects in an elastic environment. Second, it details how to support variable object content with reliable and efficient metadata management. Third, it describes special configurations to support memory-mapped programming-language objects.

2.1 Scalable management of distributed object regions

Distributed applications have diverse requirements to access and manage storage. Some characteristics for the use of storage are the frequency of accesses, the size of accessed data, the ratio of updates versus retrievals and the ratio of modifications versus creations of objects. In order to have a broad application domain, a storage service commonly defines a generic programming interface for use by different

applications. The internal structure of a storage service is determined by the definition of the programming interface as well as by application's non-functional requirements. After defining a generic interface for a storage service, this section discusses approaches to structure distributed storage in an efficient and reliable way.

2.1.1 The interface between application and storage service

A *distributed storage service* consists of storage software and physical media that is potentially distributed over multiple locations. A participant of a distributed system providing storage media is called a *storage node*. Without loss of generality, this chapter assumes each application that uses the storage service to execute on a storage node. Data replication ensures the availability of all data at each node (see Chapter 4).

An *object-based storage service* is a storage storage service that allows the application to partition storage into *objects*, chunks of memory whose contents can be defined freely by the application developer [19]. Applications access information in an object-based storage service over a NoSQL interface. A different type of storage service are distributed filesystems. While this chapter focuses on an object-based storage service, an example for a distributed filesystem based on object-based storage is given in Chapter 7.

A storage system's *application programming interface (API)* defines how applications can access and modify data. Actual storage systems implement diverse APIs such as dynamic memory allocation [22], filesystem interfaces [136] or key-value stores [60]. Some object-based systems consider objects as perennially existing, others assume unlimited storage size, such that objects may be created but need not be released explicitly. In contrast, this work focuses on *dynamic objects*, that is, objects that can be created and deleted during runtime. The implementation of dynamic objects requires the storage service to maintain information about allocated and free storage regions as part of the object metadata. Aggregating adjacent objects to regions that are handled conjointly reduces the storage required for metadata, because allocation structures are needed not per individual object, but only per region. Furthermore, considering the reduced communication overhead concerning metadata, region-based handling improves the scalability of metadata management. Loss of metadata is fatal, because it precludes the storage from accessing allocated objects and from allocating in free fragments of the lost storage. The allocation and deallocation routines must be efficient for different use cases. They must allow both the allocation of large objects as well as frequent creation and deletion of small objects.

A *variable object* is an object whose content can be updated by the application. Management of variable objects in a static distributed environment has already been described several decades ago [4, 19, 47, 127]. A plethora of more recent publications discusses elastic storage systems for constant or weakly consistent objects [88, 122, 178]. However, supporting dynamic variable objects in an elastic environment is an ongoing research topic. For example, Agrawal et al. note a tension between hard consistency requirements such as atomic multi-key updates and frequent failures in elastic systems [8]. More formally, a storage in an elastic context with frequent failures cannot fully support consistency and availability according to the CAP theorem [89].

The API of a distributed storage for dynamic, variable objects provides two classes of functions: those that determine the lifetime of objects, and those that access and modify object content. To create an object, the application calls an API function called `alloc`. During the lifetime of an object, the application retrieves and updates the object's content using `read` and `write` API functions. To destroy an object and release the storage reserved for it, the application calls an API function called `free`. Chapter 3 presents further API functions to control the consistency of stored objects.

Table 2.1 summarizes the names, signatures and semantics of the API functions performed on objects. The `alloc` function takes the requested size of an object as a parameter. It can internally reserve more storage than requested, for example to store additional metadata. The return value of the `alloc` function is either an identifier for the newly created object or a special value that identifies a failure to allocate an object. The `free` function registers the supplied object as no longer used by the allocation, such that a subsequent object allocation can reuse the destroyed object's storage. The `read` function

name	signature	semantics
`alloc`	`size→object`	create and initialize an object of specified size
`free`	`object`	destroy the object and release the storage associated with it
`read`	`object,offset,size,buffer →nread`	read a data chunk of specified size from the object into the buffer, starting at the supplied offset in the object, and return the number of bytes read
`write`	`object,offset,size,buffer →nwritten`	write a data chunk of specified size from the buffer into the object, starting at the supplied offset in the object, and return the number of bytes written

Table 2.1: Abstract API functions related to dynamic objects

extracts data from the specified object, starting at the given offset, into the buffer supplied by the application. The `write` function modifies the object at the given offset and sets it to the content of the buffer. Both `read` and `write` transfer at most the number of bytes specified by the `size` argument, so the application can use a buffer of this size or larger.

2.1.2 Storage service architecture

This chapter describes an in-memory storage that replicates objects to all participants of the storage in contrast to a partitioned database, in which the participants store disjoint parts of the information. Therefore, the replicated in-memory storage enables each application program to directly access every object that exists in the system. A straightforward approach to making all objects accessible is to colocate the storage service with the application process as a shared library. With this approach, each node contributes his memory to the overall storage capacity, such that there is no need to distinguish clients and servers of the service. Figure 2.1 exemplifies a library-based replicating storage system. The colocation reduces the communication latency between storage service and application program, but the lack of protection requires mutual trust between library and application. The aspect of fault tolerance is discussed in Section 2.2. Although objects are accessible on each storage node, a node needs not store up-to-date replicas of each object. Chapter 4 presents advanced replication policies.

The storage service API described above suggests handling object allocation and object content in separate modules within the storage service. To provide for elastic object handling, the object allocation must in turn be based on a flexible assignment of objects to nodes. The handling of object content can further be structured into an object access layer, which implements the access API, an object replication and a consistency management module. Chapter 4 argues that the latter functionalities should be implemented as separate modules.

Figure 2.2 shows a generic internal structure of an object-based storage service consisting of three layers. The bottom layer assigns each object that can possibly exist in the system a manager node. The second layer differentiates between allocated and free objects, and the third layer coordinates replication, consistency and accessibility of allocated objects. Each layer of the service provides a fundamental abstraction, such that neither lower nor upper layers are concerned with handling the abstraction. The lowest layer defines object managers, the middle layer handles object reservations, and the upper layer deals with object content.

Although objects are replicated at the upper layers, the bottom layer ensures that there is exactly one manager node for each potential object. Separating object allocation from the partitioning of the object range simplifies the consistent handling of object metadata during allocations and releases of

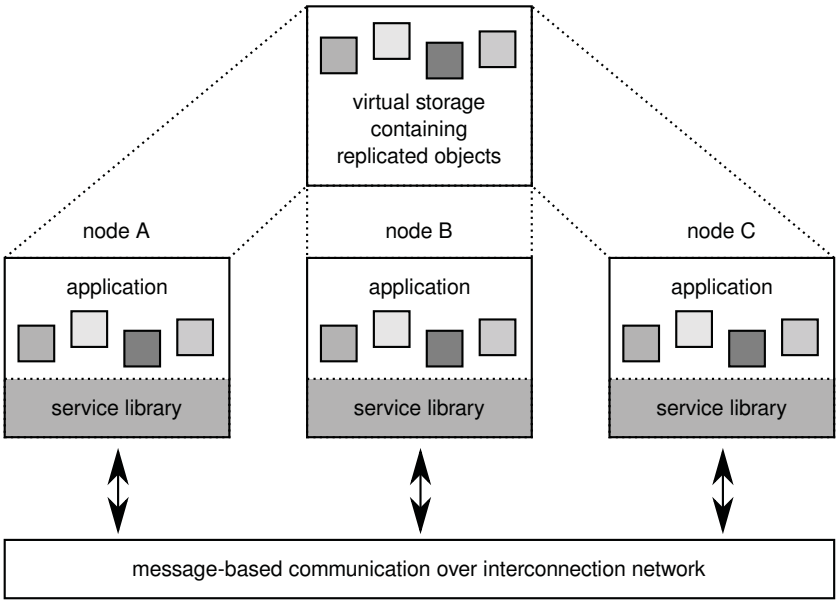


Figure 2.1: Architecture of a library-based replicated storage system

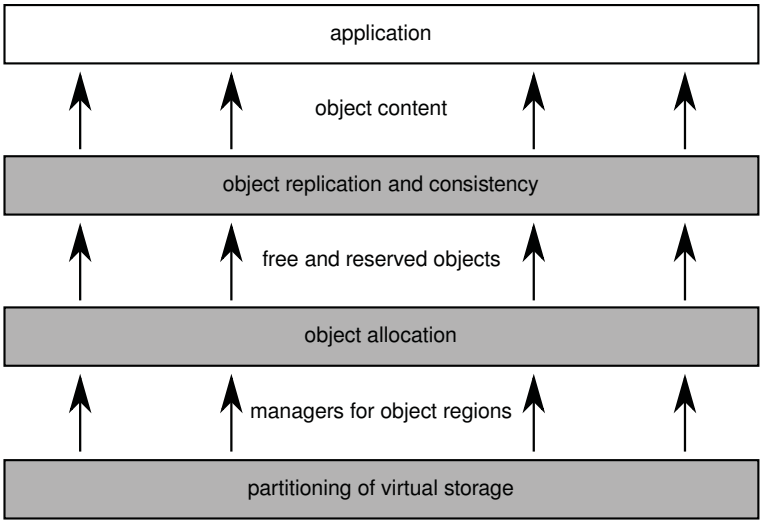


Figure 2.2: A layered storage service

objects. This chapter focuses on the two lower layers, whereas the subsequent chapters detail the upper layer.

2.1.3 Objects and manager nodes

The object allocation layer must support the object access layer with two elementary operations. First, the creation of objects needs to find free storage for the requested object. Second, operations on existing objects need to retrieve data associated with the object. Both operations must perform efficiently in the presence of many objects and nodes even in case of concurrent operations and nodes frequently joining, leaving or failing. The base operations work on distributed data structures, and their efficiency depends on the functionality that the object partitioning layer provides to the object allocation layer.

Object identifiers

When accessing or deleting a particular object, application and storage system need to agree about which object to operate on. To establish a mutual accordance among participating nodes about the object identity, an object-based system needs to define an object identification scheme. Therefore, the `alloc` function returns an *object identifier (OID)* that uniquely tags the object just created. The application uses the OID when referring to the object, such that the storage system knows on which object to perform the requested operation. The OID is also specified when destroying the object using the `free` function.

The application uses OIDs to instruct the storage system, but it does not associate any semantics with them. Therefore, the storage system can hand out arbitrary OIDs. However, the method of selecting OIDs determines important storage properties such as scalability in the number of nodes and objects, efficiency in runtime and storage requirement as well as resilience in case of failures. The *OID space* is the range of possible OID values. With an OID width of n bits, the OID space accommodates at most 2^n different objects. The amount of information that a distributed storage can host depends on the OID width and on the size of objects, which is generally bounded by the width of the addressable offset within an object. Special storage configurations, such as support for programming language objects, restrict the size of atomic objects to one byte, as will be discussed in Subsection 2.3.1. A distributed storage consisting of one-byte atomic objects is 2^n bytes large. Given that current server processors implement 64-bit addressing, allowing to address 4 exabytes of memory, 64 is a reasonable minimum width for OIDs.

OID management takes care of OID allocation for dynamic objects. A system for OID management must fulfill a number of requirements. First, OID management must support on-demand allocation of objects. To enable collocation of related objects and memory-mapping of large objects (see Subsection 2.3.4), it should support allocating ranges of OIDs. Second, the protocol must guarantee the uniqueness of OIDs, such that at any point in time, each OID designates at most one object. Therefore, the distributed information about free objects needs to be kept consistent among storage nodes. Third, to achieve high scalability in the number of participating nodes, an elastic OID management system must adapt to nodes joining and leaving during runtime, as well as to unexpected node failures and interconnection network errors. The effect of nodes joining and leaving a distributed system is called *churn* [164]. In case of a node failure, the system must not lose information about allocation status and data related to objects. Section 2.2 details a failure model for OID management. Forth, the system should handle resources economically. For example, despite guaranteeing uniqueness of OIDs, it should use the OID space efficiently, and it should process requests to create or free objects with reasonably low latency.

Node identification

The decentralized nature of a distributed storage requires nodes to communicate over a messaging infrastructure. Therefore, a node wanting to send a message to a peer or group of peers needs to be

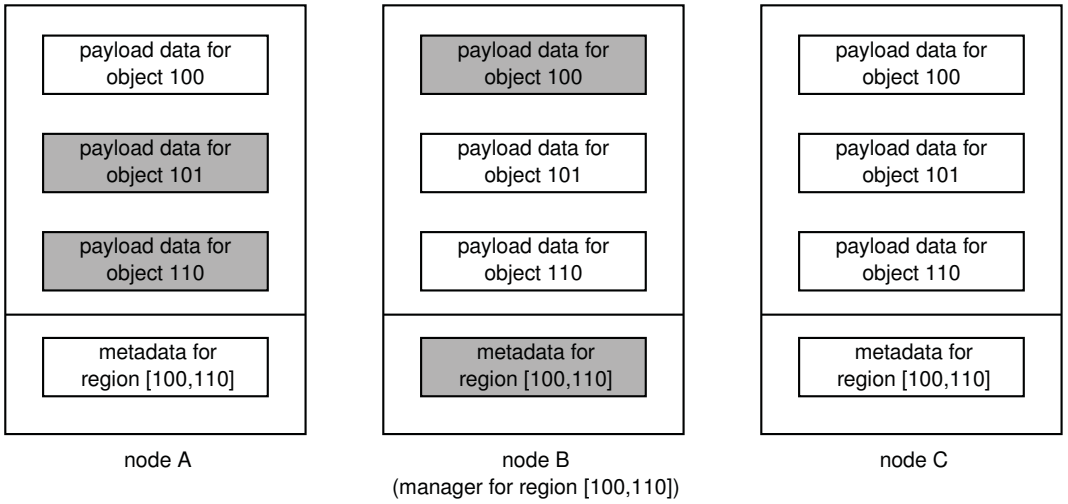


Figure 2.3: Nodes, metadata and payload data

able to address the receiver. A *node identifier (NID)* represents a computing node in a distributed system. Although inter-node communication is possible even without addressing NIDs, for example using broadcast or anycast [138], addressing the recipient using a NID is more efficient for the communication patterns discussed below.

The distributed system has two fundamentally different options to define NIDs. On the one hand, it can derive NIDs from preexisting IDs such as IP addresses, machine identifiers or time stamps. On the other hand, it can assign NIDs from a range of possible values. With the first option, the storage system uses a cryptographic hash function to calculate NIDs that are virtually unique. The second option allows the storage system to choose NIDs according to its own plan, for example to simplify associating OIDs with NIDs, but it has to ensure uniqueness of NIDs by itself. Both options use a *bootstrap node* to help a joining node, who is not yet associated with a NID, obtain a NID and integrate into the running system. The allocation and routing protocol described in Subsection 2.1.4 contains a bootstrap mechanism that assigns NIDs favoring locality for object allocation.

Object managers

Using the OID and NID concepts, a distributed system can establish nodes as *manager* for subsets of objects. A manager node takes several responsibilities. First, when the application wants to modify an object's content, the replication scheme may require to contact the manager node. For example, the primary-copy and multi-primary replication schemes use this convention (see Section 4.1.2). Second, when a node accesses an object it has not yet encountered, it needs to contact the manager for a given object to retrieve information about the object. In general, the manager needs not hold the object's primary replica, but is must at least know the object's allocation status and a replica holder from which to retrieve further information. Third, by defining object ownership, an object store can benefit from locality during allocation of dynamic objects. In case an object's previous manager has failed, the newly designated manager needs to recover the object and gather the object's distributed state.

In order to handle objects in a structured way, a distributed storage system distinguishes between *metadata*, which describes state relevant to the storage system internally, and *payload data* or simply *data*, which is accessible to applications but not interpreted by the storage system. Figure 2.3 illustrates the interrelationship of nodes, metadata and payload data. Filled boxes represent primary replicas, whereas transparent boxes represent secondary replicas. By convention, the primary replica of the distributed metadata is always located at the manager node (see Subsection 2.1.4). The location of object's primary replicas is variable, such that the storage service can move primary replicas to the nodes that access the object frequently.

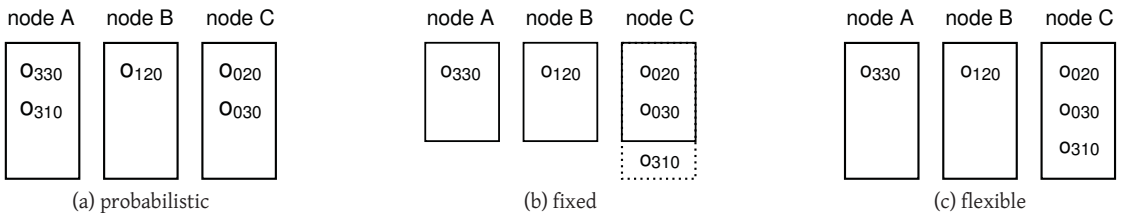


Figure 2.4: Mapping of objects to manager nodes

Association of objects and manager nodes

To associate objects with manager nodes, it suffices to consider the association of OIDs and NIDs. The storage service can select both kinds of identifiers freely, and choose the identifiers such that object creation and lookup are scalable and efficient. There are two fundamentally different approaches to generate OIDs. On the one hand, the storage service can take a probabilistic approach such that identifiers are scattered over the possible ranges. On the other hand, it can establish a mapping of OIDs to NIDs, which enables better control over the distribution of IDs.

Probabilistic identifiers Management of OID space can make use of probabilistic theory to guarantee the uniqueness of OIDs almost certainly. *Distributed hashtables (DHTs)* generate probabilistic OIDs by computing hash keys of object attributes such as human-readable names or static object content [156, 169, 178, 197]. An advantage of DHTs is that an object can always be found in a stable system without any central component. Participating nodes only need to know the hash function and a subset of their peer nodes, whose cardinality is most commonly a logarithm of the maximum number of nodes. By exploiting consistent hashing [110], the DHT needs not relocate all objects in case of node churn. The range of NIDs is selected to equal the range of OIDs, such that nodes are responsible for a certain subset in the OID space which is proximate to their own NID. The communication mechanism used by DHTs is called *key-based routing* [150], because, when searching for an object, its identifier encodes the route to the node storing the object. Whenever a node receives a search request for an object it is not responsible for itself, it forwards the request to a peer whose NID is numerically close to the searched OID, such that the request finally arrives at the appropriate node. If the subset of peers is chosen appropriately, lookup operations require only logarithmic time in the number of nodes in the system. For example, the Chord DHT uses finger tables that contain pointers to nodes with exponentially growing distance [178]. Figure 2.4a exemplifies OID mapping of five objects to three manager nodes using a DHT. A possible hash function for the example is $h(d) = \sum_{k=0}^{n} d_k \pmod m$, where $d = \sum_{j=0}^{n} 2^j$ is a positional notation for d with $n + 1$ digits, and $m = 4$ is the expected maximum number of nodes in the system.

Although DHTs scale well and enable efficient routing for large numbers of objects in presence of node churn, the commonly used consistent hash functions conflict with locality of reference. Hash keys of similar-named objects usually diverge and are mapped to different nodes in most cases. DHTs using locality-preserving hashing have been proposed [82]. Object allocation based on locality-preserving hashing would still an lack efficient method to find unused ranges in the OID space.

DHTs partition the object ID space depending on the IDs of the participating nodes. Therefore, the ID space is often unevenly partitioned, such that network traffic and storage load is often misbalanced [111]. Concepts such as virtual nodes [178] and item balancing [111] relieve the misbalance, but the problem is inherent to the combination of key-based routing and probabilistic IDs. Furthermore, if OIDs are not wide enough, choosing IDs in a probabilistic manner is not collision-resistant, because the chance of duplicate hash values would be too high [153].

The key-based routing approach works very efficient if object content remains constant and objects are accessed independently of each other. The multi-hop routing does not take any precautions for object updates. For example, network restructuring can cause messages to take different routes, possibly overtaking one another.

Fixed mapping of identifiers to nodes A second approach for defining OIDs is to encode the manager node in the OID, such that the manager's NID can be calculated from a given OID as a local operation. Figure 2.4b applies a fixed mapping to the same objects and nodes as in the previous example. The manager node for an object is determined by removing the OID's last two digits.

Given an OID, the fixed encoding allows to determine the manager node in constant time, independent of the number of objects and the number of nodes in the system.

However, the fixed-encoding approach numerically limits the number of nodes and the number of objects per node. For example, when using 16 Bit NIDs and 32 Bit OIDs, there may exist at most 65536 objects per node.

Another drawback of the fixed-encoding approach is that its resilience is limited. If a node leaves or fails, the objects managed by it become orphans. A work-around would be to let a peer become responsible for the objects managed by the failed node. Given that the peer will have a different NID, object lookups by other nodes would fail until every node in the system has noticed the ID change.

Furthermore, the fixed-encoding approach does not allow for load balancing, because nodes are not able to offload management of objects to peers that are less busy.

Flexible mapping of identifiers to nodes An object store that contends neither with probabilistic identifiers nor with a fixed mapping of objects to nodes needs to allow the assignment of objects to nodes to change over time. Figure 2.4c shows a flexible mapping of objects to managers.

In contrast to the previously outlined approaches, flexible mapping requires metadata to store information about which manager is responsible for an object. Flexible mapping must store all metadata explicitly and make it available to all nodes, because, unlike with key-based routing or fixed encoding, the manager node's NID cannot be derived from OIDs implicitly.

With a mapping scheme where an object can be mapped to any node, the number of messages needed to locate an object is generally linear in the number of storage nodes. In the worst case, a node needs to contact all his peers. Routing in time linear to the number of participating nodes is unacceptable for a scalable system. To increase efficiency of routing requests to flexibly placed objects, a storage system can apply a number of techniques. First, the flexible mapping can restrict the group of nodes that can an object can be mapped to. For example, by mapping even OIDs only to nodes having an even Node ID, and odd OIDs only to nodes having an odd Node ID, the storage could half the maximum number of nodes to ask for an object. The disadvantage of this approach is that it not only restricts the possible mappings, but also the flexibility of the storage system. Second, replicating the mapping tables enables nodes to make local decisions about which peer to contact for a given OID. For example, if each node has a local copy of the mapping tables, it can directly contact each peer without ever needing to send a message over several hops. In practice, keeping all tables consistent would require a high number of network messages. Third, a best-effort approach can combine the previously mentioned techniques to a flexible solution with good average-case performance. The system should allow finding nodes with the help of key-based routing, and its should partially replicate mapping tables to guarantee availability of mapping information. The following subsection details the implementation of an OID space partitioning system supporting flexible partitioning of the OID space and key-based routing of messages to object managers.

2.1.4 Key-based routing for fast object retrieval

The two fundamental operations that a storage service needs to implement dynamic distributed objects are finding free storage regions and retrieving the manager corresponding to an OID to access data and metadata. Based on the definitions and insights from the previous subsections, the allocation and routing protocol described in the following combines key-based routing towards manager notes with the option to select OIDs in a non-probabilistic way for locality-preserving allocation.

Identifier mapping protocol

Key-based routing enables efficient object lookups in an elastic environment. However, instead of generating probabilistic OIDs using consistent hashing, the proposed protocol supports dynamic allocation and releasing of OIDs. In contrast to DHTs, the protocol takes care of load distribution itself in a manner that minimizes fragmentation and tolerates node churn. To establish a simple relationship between objects and manager nodes, it selects NIDs from the same range as OIDs.

The allocation protocol tries to allocate objects close to the requesting node. As a means to achieve locality among objects, the protocol aggregates OIDs to OID regions. Due to varying storage consumption, ranges of the OID space can sometimes be entirely occupied, so that free storage is available not at the allocating node but only at nearby nodes. In these cases, the system can satisfy an allocation only after reassigning OID regions from nearby nodes. To handle nodes joining and leaving during runtime, a stabilization protocol similar to the ones used by DHTs maintains the key-based routing structure. Each participant activates the stabilization protocol in regular intervals to update the entries in its routing table.

The first node starts the system by selecting an initial NID and creating the first object region. The first region represents the whole OID space of $(\text{offset}_0, \text{size}_0)$. Allocations result in the first region being split into smaller regions.

A node that joins the running system contacts a well-known bootstrap node and requests a NID. To facilitate future allocation of OIDs by the joining node, the bootstrap node assigns it a NID that resides in a large free OID region. If the bootstrap node does not have a free region large enough itself, it may need to locate a large region at another node first. (see Section 2.2).

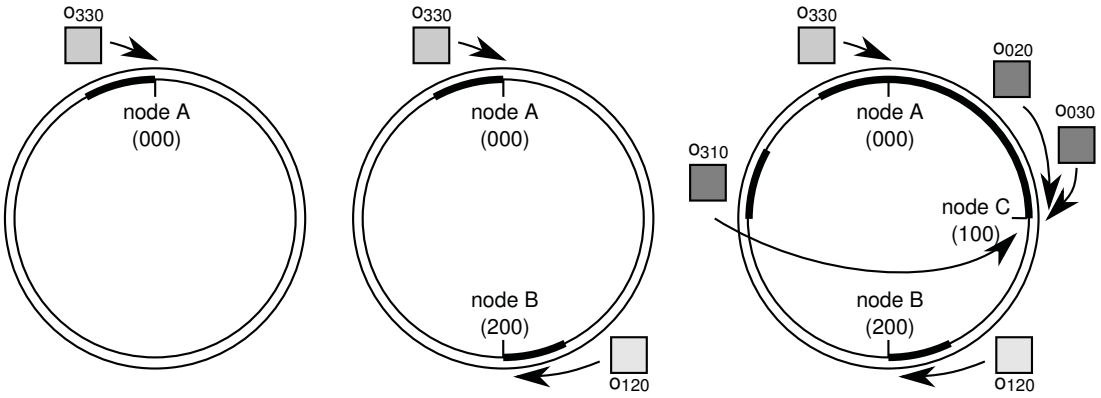
Nodes generally satisfy allocation requests only from OID regions assigned to themselves. Whenever a node wants to allocate an OID region, it tries to find a free region already assigned to it. If it finds an appropriate region, and the found region is larger than what is requested, it splits the region, returns the matching region and keeps the remainder. If the node does not have a region large enough, it requests a free region from its peers. To this end, it sends a *map request* to its predecessor node in the OID space. The map request is forwarded until it reaches a node which still has a free region that is large enough. The found node then chooses a region it owns and transfers ownership to the allocating node, who can in turn finally proceed with the allocation. The options for selecting a region to remap will be discussed below.

Requesting OID regions from predecessors in the OID space has two advantages over requesting them from successors. First, subsequent searches for the object will be routed to the predecessor, who then knows how to contact the manager node directly, because it has handed the corresponding region over to the manager before. Second, in case the predecessor quits or fails, the stabilization protocol ensures that search requests are routed to the object's manager node automatically.

Figure 2.5 displays the evolution of a mapping, beginning with the initial node A having allocated one object (Figure 2.5a). A second node B joins the system, obtains half of the ID space and creates another object therein (Figure 2.5b). The third node C allocates regions for two objects but then runs out of space for a third object, so it requests additional object identifiers from its predecessor, the initial node (Figure 2.5c).

Identifier lookup protocol

The identifier mapping protocol outlined above works towards efficient lookup of manager nodes for objects. When searching for object content or metadata, a node has to contact the manager for the relevant OID. The lookup protocol sends request using key-based routing over several peers towards the manager. The key-based routing proceeds similarly to routing in a DHT and works in a fully distributed manner. The routing of a message concerning an object is based on the object's OID only. Nodes do not keep full routing tables for all their peers. Instead, they forward a message to a peer they know whose ID is numerically closest to the target OID until the message arrives at the responsible manager. The forwarding always terminates, because each hop reduces the distance to the manager by at least one step in the ID space.



(a) initial node (b) two nodes (c) three nodes

Figure 2.5: Identifier mapping

Routing tables such as Chord's finger tables [178] help cut down the number of hops to a logarithm of the number of nodes. In a finger table, a node stores peers that are approximately a logarithmic distance away in the OID space. For a node having NID n, the first entry is its direct successor, the second entry is the node responsible for key $n + 2$, the third entry is the node responsible for the interval $n + 3$ to $n + 4$ and so forth. When a node receives a search request for an object it does not manage itself, it forwards the request to the node indicated in the finger table. Like in Chord, the number of entries in a routing table should equal the number of bits in an identifier. Other key-based routing schemes can be used alternatively.

Dynamic resource consumption can cause an object to be mapped to a manager other than the node associated with its ID. Assume that in Figure 2.5c, the bootstrapping protocol has assigned node A the ID 000, B the ID 200 and C the ID 100. Object o_{310} in Figure 2.5c is managed by node C, although naive key-based routing would end at node A, which is numerically closer. Therefore, node A, which knows that node C manages o_{310}, must pass on the lookup request to C. In case the manager is further away from the OID, replication of mapping metadata ensures the traceability of the manager, as will be discussed below. The system can avoid endless forwarding loops by limiting the number of recursive forwarding hops and retrying lookup operations iteratively. Figure 2.6 shows two alternative lookup routes for object o_{030} starting from node B. The solid arrow in Figure 2.6a illustrates that, if B knows that the object is managed by C, it sends its request directly to C. The solid arrows in Figure 2.6b show that, otherwise, B sends its request to A, who forwards it to C.

Scalability and performance

To be able to reason about scalability and performance of OID allocation and lookup, some assumptions about the application environment of the storage service are needed. As said above, the OIDs are at least 64 bit wide. Although objects are allocated in chunks, the OID space will be sparsely populated. The number of participating nodes may be in the magnitude of several thousands, but still much less than the number of objects.

Based on the assumptions about the application environment, the storage system can optimize the remapping of OID regions with respect to their OID and size. To retain locality among objects and nodes, the donator of a free region should select a region having an OID numerically close to the NID of the receiver. In case the donator owns large unused regions, it should over-provision the receiver. Mapping regions larger than required avoids the receiver having to request further regions in the near future.

The search for free OID regions can further be improved by using a second kind of finger tables. Each node keeps a table that stores, for different region sizes, the nearest neighbor having a free region of the respective size. If a node wants to allocate storage and does not have enough free storage by

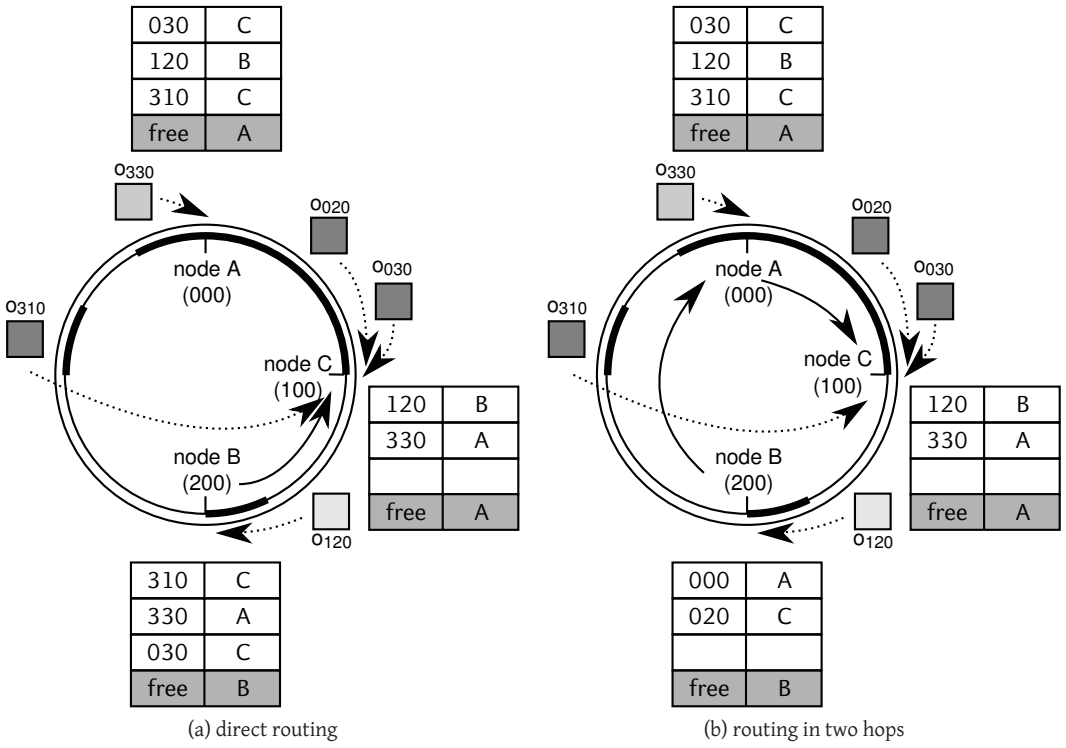


Figure 2.6: Identifier lookup

itself, it searches for a free storage region with sufficient size. Usually, it will start its search with its direct predecessor, so that the association of objects and manager nodes preserves locality, and the storage does not fragment unnecessarily. However, if the neighbor has also run out of free storage, the lookup must proceed to nodes further away. Without a secondary finger table, the search would require iterative or recursive messages. Secondary finger tables help reduce the communication by allowing the lookup queries to take shortcuts. It does not cause any harm if some lookups do not find the closest matching regions, as long as the majority of lookups preserves locality well. The secondary finger tables must be updated regularly much like the primary finger tables used for message routing by searching for free storage regions in power-of-two sizes.

The structural deviations from DHT-like implicit OID mapping necessitate handling those cases where naive key-based routing does not succeed in finding the manager node. Linearly searching the participating nodes would require a high maximum hop count to avoid frequent delivery failures. Therefore, the mapping information should be replicated at all nodes between the original manager, which is found by naive key-based routing, and the current manager. The replicated mapping information takes a function similar to the leaf sets in Pastry [169] and the neighbor links in Tapestry [197]. Pastry's neighborhood set contains nodes that are physically close, such that the efficiency of locality-driven searches is increased.

Mapping information must be kept consistent in order to avoid failing searches. In contrast, replicated metadata needs not always be up-to-date, because, for each object, the manager node itself synchronizes metadata. Outdated mapping information can cause messages to be routed over detours, or in the worst case a message is discarded after exceeding the maximum hop count, such that it does not reach its receiver. Given that mapping information changes only gradually, outdated mapping information occurs seldom. Failed lookups can be handled by repeatedly sending the request, because the system will eventually have consistent routing tables. If a message is discarded, the requesting node will resend it after a timeout, and the mapping information should have stabilized by then. In case a node

leaves the system during runtime, it remaps its regions to its peers. Section 2.2 describes a protocol to transfer the ownership of a region between nodes in case of a failure.

Downsizing fragmentation

A space-efficient allocation scheme must not excessively waste OID space. Unusable space inside of allocated regions is called *internal fragmentation*. Internal fragmentation occurs if the regions assigned are larger than the requested size, for example if the allocator pads regions to obtain a minimum size or a multiple of a fixed size.

Unusable space between allocated regions is called *external fragmentation*. External fragmentation is usually caused by small regions being deleted while neighboring regions remain allocated. Identifiers such as OIDs must remain constant once allocated, thus they must not be relocated within the OID space. Therefore, external fragmentation is an important issue with long-running allocators.

The buddy system is a well-known allocation scheme, which counteracts fragmentation and is simple and efficient to implement [116]. Although the buddy system does neither preclude internal nor external fragmentation, it restricts internal fragmentation to below half of the amount of allocated space, and usually causes little external fragmentation. In the buddy system regions always have a power-of-two size and are aligned to their size, such that they can be represented in the *flexpage* format [128]. The system allows to half a region into two smaller regions. Only regions that have once been split are considered as buddies that may be coalesced later.

The buddy system can help structure the OID space to limit internal and external fragmentation. The system goes well with preserving locality, and its simplicity facilitates a straightforward implementation. Storing regions of the same size in a sorted data structure such as a B-tree [20] reduces the time required to find a buddy whose OID is close to the NID of the requesting node.

2.2 Reliable metadata

A distributed storage system manages global state and simulates a shared storage. Therefore, all distributed information exists as virtualized data on the participating nodes. In order to provide storage objects for use by applications, the storage system needs to maintain metadata about these objects.

Participants of a distributed storage system need to coordinate their actions. To this end, they store *metadata*, that is, non-payload information about objects for use by the storage system itself. In general, there exist two kinds of metadata for distributed objects: some metadata is shared among the nodes similarly to the payload data, whereas other metadata is kept and meaningful only at a specific node.

2.2.1 Shared metadata

Shared metadata comprises information about the partitioning of the OID space. The partitioning information changes when the storage service splits or merges adjacent non-allocated object regions. Besides, distributed storage needs to keep certain object properties. For dynamic objects, the metadata must keep track of whether any storage for the object is allocated. The allocation state changes when the application calls `alloc` or `free`. Allocated objects can have further properties that are stored in metadata, such as the consistency model an object is bound to, default values or cleanup routines to destroy objects during deallocation.

Shared metadata is initialized or updated when the storage service allocates an object. In most cases, metadata remains constant after allocation. However, to enable nodes to access metadata stored on a peer, the metadata management must provide an interface to retrieve and change object properties.

In the three-layer design presented in Subsection 2.1.2, the top layer of the storage system replicates payload data to make it accessible on each node. In contrast, the bottom layer of the system partitions metadata among the nodes. Partitioning of metadata means that each object is assigned to exactly one manager node. This convention allows a specialized and more robust implementation of metadata handling, because it simplifies the handling of metadata consistency and resilience.

In a large-scale distributed system, a single node cannot be considered to be reliable online, because software bugs, node or network failures can happen, and with a large number of participants these hazards are very probable. To guarantee resilience in case of failures, the storage service cannot rely on individual manager nodes. Therefore, it must cache shared metadata in secondary read-only copies. Metadata is variable, although it does not change as frequently as payload data, so the storage service requires a consistency model for metadata.

Using different consistency models for data and metadata can lead to race conditions. For example, creation of an object initializes its metadata, but modifying the object shortly after its initialization may already alter its metadata. The storage service must ensure that data and metadata consistency are compatible. In practice, the service can assume that payload is modified much more frequently than metadata. This allows to optimize the metadata management for reliability rather than for performance.

All distributed metadata is bound to a specific object, and metadata of different objects is mutually independent. Therefore, it suffices for a storage system to guarantee coherence with respect to an individual object. As long as an object's manager node remains the same, it can hold the primary copy of the shared metadata and serialize access to it. A node requiring up-to-date metadata can always contact the manager node. However, if he accepts metadata that is potentially slightly outdated, he can access secondary replicas. The primary metadata copy is always located at the manager node, such that all metadata updates must pass through the manager node. This does not limit scalability of metadata management, because metadata modifications are usually requested by the manager, and metadata accesses are less frequent than payload accesses. For example, the allocation of an object is a local operation in the frequent case and does not require communication with other nodes, except for updating the secondary replicas. The identifier mapping protocol presented in Subsection 2.1.4 works towards locality during object allocation.

An object's manager node synchronizes access to its metadata. Peer nodes can satisfy metadata retrievals from secondary replicas, whose content can sometimes be slightly stale. In case a peer needs to access the current state, it can directly contact the manager node. All metadata operations except for the merging and splitting apply to exactly one object. The merging and splitting operations apply to non-allocated regions only, and the buddy system ensures that both types of operations work on adjacent regions. A single manager serves as the synchronization point for these operations, such that the metadata management needs not implement a sophisticated inter-object consistency model.

It is crucial to define safe default values for metadata and data. The initial state of local metadata must be well-defined, because access to distributed metadata can take much time due to communication with remote peers. Therefore, the metadata management must define default values that are used unless more specific information is available. An example for default values are probable manager nodes that are initialized based on key-based routing information. The initial metadata lookup can adapt this information if it finds another manager holding the metadata.

Several optimizations simplify the handling of metadata. First, it is reasonable to define a special identifier for default object content. A single identifier can represent an object that is fully zero-filled, such that the storage overhead and data transfer time are reduced for initial and frequently seen state. Second, requests and replies for an object should piggyback all known metadata. Piggybacking avoids race conditions between data and metadata requests (see Subsection 2.2.4), and it has the additional advantage that metadata automatically propagates to nodes possibly interested in it. If a node does not yet know metadata properties, it should initialize them with a special marker for being *undefined*, such that it can update them with defined values when receiving metadata from a peer later. Third, the storage service must guarantee the availability of metadata even after an object has been deleted as long as references to the object exist. Thus, it should keep metadata for a back-off period after object deletion. The deletion of an object is partly related to consistency and replication management. Therefore, Subsection 3.1.2 discusses the aspects of object deletion that are related to replica management, and Subsection 4.2.3 the aspects related to storage consistency.

component	type of metadata
consistency control	object accesses per access sequence
storage replication	queue of versions per replicated object
OID space partitioning and object allocation	cached distributed metadata

Table 2.2: Components of the storage system which keep local metadata

2.2.2 Local metadata

Each node needs to keep track of the objects it encounters during the run of a distributed application. The state stored includes coherency and consistency state, information about object replicas at other nodes and references to data content. Table 2.2 summarizes which components of the storage system keep local metadata.

Caching of distributed metadata is an important purpose of local metadata. Keeping local information about frequently accessed objects makes them accessible at a node without needing to communicate with a remote peer. In addition, local representatives for distributed objects enables the storage to keep resources that are expensive to create and delete over a longer time period. For example, the storage can retain a memory mapping of an object (see Subsection 2.3.1) so it does not need to create and destroy the mapping each time the application accesses the object. Cached metadata is usually created when accessing or receiving any information related to an object. The system must not lose distributed metadata, so that a certain number of metadata replicas always exists. Other cached metadata may be discarded at any time.

The scalability of consistency protocols benefits from aggregating object accesses to access sequences. In the course of an access sequence, the storage system gathers metadata about individual object accesses. For example, it must store payload data written to the object, and information about the object's state at the time of access. If the number of objects accessed is small compared to the total number of objects known locally, access metadata must be stored in a dedicated data structure.

Information about replicated objects is also kept using private metadata (see Chapter 4). For each object that is replicated at a node, the system must store payload data and further information. A multiversion storage system (see Chapter 3) does not have the notion of a *current state*. Instead, each object has a queue of versions, each storing payload data and additional metadata.

The local metadata state must match the distributed state. However, the synchronization of metadata is not as complex as the synchronization of payload data (see Chapter 3). An appropriate synchronization strategy precludes most potential race conditions. For example, when a node accesses a newly created object, it must access the metadata at first in order to determine the object's manager. Accessing objects that are being deleted is free of hazards if subsequent allocations do not instantly reuse the same OID.

2.2.3 Fault-tolerant metadata management

In case of failures, the storage must guarantee the availability of metadata. It must neither lose information nor access inconsistent metadata after a failures. The failure model of this section assumes fail-stop of single nodes.

The metadata describing the partitioning of the OID space requires special care. If partitioning of the OID space is handled in a decentralized manner, as described in Section 2.1, the manager node serializes metadata updates. Therefore, modifications in partitioning information are critical for the consistency of OID space partitioning.

Tolerating node churn

As mentioned above, a large-scale storage service must be able to handle nodes joining and leaving the system efficiently. The integration of a joining node has already been described. If a node voluntarily leaves the system, the object regions assigned to it have to be remapped to its neighbors. To take advantage of already replicated metadata and to accommodate key-based routing, the successor on the ring should take over the object regions, unless it is already heavily loaded.

As long as metadata remains at the same node, the metadata replica holders ensure the recoverability of metadata. The transfer of region ownership requires special attention to avoid losing in-transit regions. The reliable transfer of a storage region can use the protocol described in the following paragraph. The protocol builds on established techniques, namely reliable communication, leadership election and a variant of two-phase commit.

Before transferring the ownership for a region to a peer, the node notifies the holders of metadata replicas for the respective region about the remapping intent. Besides sending the identifier and size of the region, it includes the intended new manager and the list of nodes that hold metadata replicas in its message. As soon as the new owner has received the region, it acknowledges the transfer to the replica holders. If the old manager fails during the transfer, the new manager finishes the transfer by gathering missing metadata from other replica holders. If the new manager fails during the transfer, the old manager rolls back the failed transfer and subsequently tries to transfer the region to another node. In the unlikely case that both the old and the new owner fail during the transfer and the replica holders do not receive a commit or abort message until a timeout, they elect a new manager for example using the Bully algorithm [86]. The newly elected manager then reconstructs the storage region.

Tolerating node failures

Failures of nodes can be handled using a protocol similar to region remapping. If a node detects the failure of a peer, it notifies the node that will take over the regions. This is regularly the successor of the node or, if that node has also failed, the node following the successor. The successor of the node then iteratively acquires ownership of all regions. It notifies the other replica holders of its transfer intent, gathers the metadata and finally acknowledges the transfer to the replica holders. As a last resort, the in-memory storage can use a disk-based checkpoint-and-restart mechanism, which can result in losing some data but at least does not risk data inconsistencies [67, 92].

2.2.4 Remote free operations

Freeing an object allocated by another node is designated as *remote free*. For a distributed storage service that establishes an association between objects and nodes, remote free requires special handling because of potential race conditions between the freeing node and the allocating node [68]. The allocation protocol described in Subsection 2.1.4 allows only the remote manager node to modify the metadata. If an object has once been allocated on a remote node, the freeing node must contact the remote manager in order to change the metadata associated with the object.

The remote free problem manifests differently for two object store configurations. On the one hand, if the assignment of objects to nodes is flexible, a remote free can be implemented as a change of storage ownership, such that the free storage afterwards belongs to the node who has released the object. On the other hand, a fixed assignment of objects to nodes requires the node freeing the object to notify the node managing the respective storage region.

The consistency model for payload data can introduce race conditions. Consider the releasing of an object that is referenced by another weakly consistent object. While the free operation is in progress, it can happen that the object itself has already been destroyed, but the reference to the object has not been invalidated yet. If a node then tries to access the object and the metadata has already been deleted, it fails to retrieve the metadata. Therefore, the metadata must be handled with a consistency as weak as the payload data. Another countermeasure against race conditions between payload data and metadata is to piggyback metadata on each operation, as already mentioned in Subsection 2.2.1.

A remote free operation is asynchronous to the node who has allocated the object. To avoid damaging allocation metadata, the freeing node must synchronize with the allocator. However, the allocator need not free the object immediately, it can register the object to be freed and update the metadata later. Considering potential race conditions, it is safer to delay freeing metadata, until references to the object do not exist any more. Implementations can furthermore use reference counting and other garbage collection techniques [131] to avoid these race conditions. Chapter 3 details the implementation of remote frees in transactional consistency.

2.3 Support for different kinds of objects

A generic API for storage access has been outlined in the beginning of this chapter. In general, the storage API can be implemented in any imperative programming language for any possible configuration of data store and hardware. A deeper integration of object-based storage and programming language can benefit access performance and ease the implementation of distributed applications. To directly support distributed programming language objects transparently with respect to applications, a distributed storage needs to implement memory-mapped objects and handling of small objects. Memory-mapping of large objects requires special attention with respect to data consistency.

2.3.1 Memory-mapped objects

A distributed storage service can make programming language objects accessible in the application's address space by mapping them into memory. Applications can access memory-mapped objects just like non-distributed objects by identifying OIDs with memory pointers. The implementation of memory-mapped objects is straightforward with a library-based implementation of in-memory storage, because objects always reside in main memory, and the storage service library can directly manipulate the application address space. However, mapping objects into memory is possible only for certain configurations of distributed storage.

Distributed storage can define the relationship between OIDs and object content in two fundamentally different ways. In the first kind of configurations, objects have variable size. The content of a variable-size object is designated by a single OID. It may be possible to change the size of an object dynamically, but regardless of object size and offset specified, it is impossible to access an object other than the one specified by the OID. In some cases, such as programming-language objects or filesystems, it is useful to have a fixed minimum size for blocks of consecutive objects.

In the second kind of configurations, all objects have a fixed size. By specifying an offset relative to an OID, an application can access the contents of other objects. Aggregating objects into fixed-size blocks benefits performance, and using offsets to address other objects simulates variable-size objects, for example to support Millipage allocation (see Chapter 5). Furthermore, fixed-size object configurations allow to identify virtual memory addresses with OIDs.

Definition 5 *A* memory-mapped object *is an object whose content is located in virtual memory at the virtual memory address corresponding to its OID.*

The basic unit of allocation in virtual memory is the memory page. Most modern CPUs support multiple page sizes, but operating systems usually have a default page size. For example, the x86-64 architecture traditionally uses a page size of 4 KB. Memory-mapped objects reside in the application's virtual address space, so the OID width of memory-mapped objects must equal the CPU architecture's virtual address width. Virtual addresses on current server CPUs such as x86-64 are commonly 64 bits wide. CPUs can address each byte of memory separately, so atomic objects are 1 byte large. OIDs of atomic objects are consecutive, so offset 1 from OID x is object x+1. The storage can detect accesses to memory-mapped objects by configuring the CPU's memory-management unit (MMU) appropriately (see Subsection 2.3.2). MMU-based access detection has the benefit that only the first access to a page in a sequence of accesses causes a page fault, whereas successive accesses are as fast as accesses to local memory [127]. Table 2.3 contrasts explicit object accesses with transparent accesses having equivalent

	explicit access	transparent access
access	`read(oid, 0, 4, &var);`	`var = *oid;`
mutation	`write(oid, 0, 4, &var);`	`*oid = var;`

`var` is a 4-Byte local variable.

Table 2.3: Explicit versus transparent object accesses

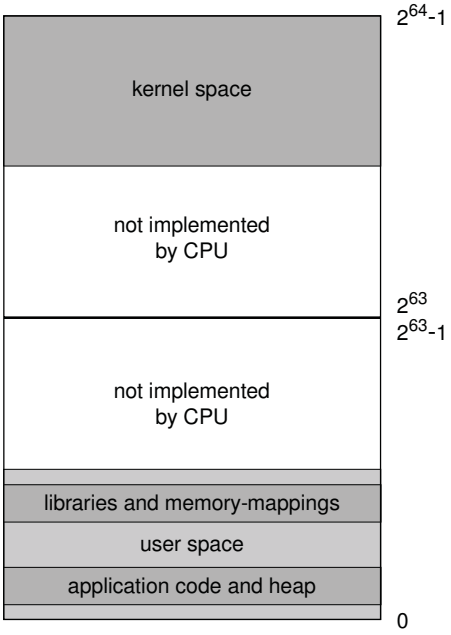


Figure 2.7: Address space of a x86-64 Linux application

effect.

The implementation of memory-mapped objects must take the actual address space layout into account. Address space layout is determined by the operating system kernel and user-level runtime libraries such as the C standard library. To achieve better performance of address translation, some CPU architectures such as x86-64 do not implement the uppermost bits of virtual addresses, but instead automatically sign-extend the uppermost valid bit [134].

The Linux kernel for the x86-64 architecture takes possession of the upper half of the address space, such that all virtual addresses in the kernel have the uppermost bits set. User-level code and data reside in the lower half of the address space. The user-level address space holds executable code of applications and libraries, memory-mapped files and dynamically created anonymous storage. In Linux, runtime libraries and kernel use memory mappings for all regions of the user-level address space, effectively pre-reserving certain ranges in the user address space. These reservations are small compared to the whole 64-bit address space. Figure 2.7 displays the address space of a x86-64 Linux application. Deviations from the default address space layout, caused by applications mapping files or anonymous storage to fixed locations, are rare. Despite non-implemented address bits and pre-reserved ranges, the user address space remains large enough to assume certain address ranges are available for distributed memory-mapped objects.

2.3.2 Hybrid access control

In order to allow concurrent operation on different objects, a distributed storage service needs to control accesses to the objects at a reasonably fine granularity. Object access control entails not only providing the correct payload data according to the consistency model associated with the object, but also detecting and registering object accesses in order to ensure the consistency of the storage.

The `read` and `write` functions can easily register accesses, because the OID of the accessed object is supplied as a parameter of the function call. Instrumentation of executable code can map object-oriented program code to `read`/`write` access control. In contrast, the storage service needs to detect accesses to memory-mapped objects, because access to memory-mapped objects is fully transparent to the application program. After describing a way to implement access detection for memory-mapped objects, this subsection presents hybrid access control that allows applications to use `read`/`write` and memory-mapped accesses side by side.

Transparent access detection

Modern processors for desktop and high-performance computing support virtual memory addressing. Virtual memory introduces a level of indirection between virtual addresses, which user-level software knows, and physical addresses, which processors and main memory deal with. By relieving applications from coarse-grain memory allocation and considerations of multitasking, virtual memory simplifies the development of applications. Moreover, virtual memory renders the existence of memory hierarchy transparent with respect to applications, such that they can access secondary storage by reading from and writing to main memory.

In many operating systems including GNU/Linux and Microsoft Windows, memory mappings are classified into named mappings and anonymous mappings. Anonymous mappings are simply designated by their memory address. Anonymous mappings suffice for simple shared memory setups, and they are simpler to handle, because they do not consume any resources in a global namespace. Named mappings allow user-level code to refer to the mapping using a name, for example a filename, a POSIX object name or a System V shared memory ID [183]. The name for the mapping enables more advanced use cases such as sharing a mapping among several processes or reattaching a mapping to several virtual addresses in the same address space. Chapter 5 describes how a storage library can use named mappings to avoid false sharing situations for MMU-based transparent access detection.

Most operating system kernels set up virtual memory on behalf of applications, such that virtual addressing is transparent to user-level software. If the operating system allows user-level software to partly configure virtual memory by itself, applications can build special features into virtual memory management. User-level reconfiguration of virtual memory enables transparent detection of memory accesses, a technique developed in the context of distributed shared memory systems [127].

In the initial state, all memory pages that are part of the distributed shared memory are inaccessible, that is, the read bits and the write bits in their page tables are unset. If the application tries to access an object on an access-protected page, the processor generates a protection fault, interrupts the regular code execution and activates a handler in the operating system kernel. The storage system library can register a user-level function that is executed subsequently with the kernel. After ensuring the consistency of the object the application is about to access, the library function grants the access rights by executing a system call to set the read and/or write bit in the page table. On UNIX and Linux systems, the system call is commonly called `mprotect`. The page remains accessible until the storage library revokes the access rights in order to avoid the application to experience inconsistencies. The revocation of access rights uses the same system call with inverse parameters. Figure 2.8 shows the state transitions that can occur during transparent access detection.

Hybrid mechanism

Access detection using MMU only can suffer from false sharing effects and frequent page faults, depending on the size and distribution of data accessed. Hybrid access control is one means to circumvent these

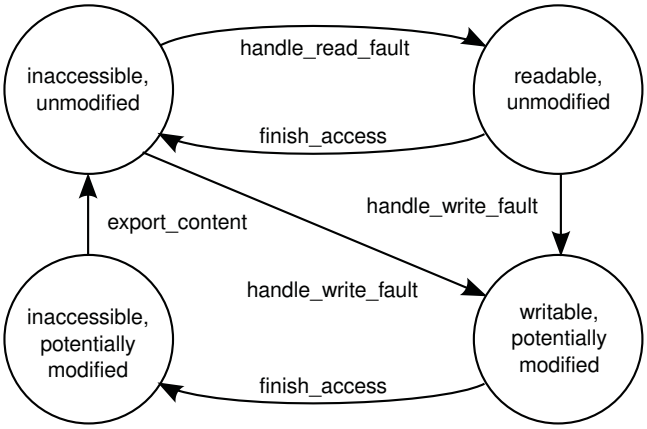


Figure 2.8: State diagram for transparent access detection

```
begin of access sequence:       read page fault:
foreach accessible page                 handle_read_fault()
        finish_access()

end of access sequence:         write page fault:
foreach modified page                   handle_write_fault()
        export_content()
```

Figure 2.9: Pseudo code for hybrid access control

problems. A storage service that provides memory-mapped objects can implement transparent access detection as an extension to explicit access control. To this end, the store comes into action to request notifications of object accesses and to handle detected accesses.

In order to receive notifications of object accesses, the data store must configure the MMU hardware to generate page faults. To request access detection for a certain address, the store must revoke the memory access rights for the memory page corresponding to the address.

When converting detected accesses into calls to the read and write functions, the CPU page size indicates the alignment and offset passed to the functions. In contrast to using explicit `read` and `write` functions solely, transparent access detection allows accessing and modifying memory after the first detected access. Once a page has been made writable, the application can modify its content repeatedly. Therefore, the store can determine the final state of a written memory page only after revoking access permission. Figure 2.9 shows pseudo-code for managing accesses. The `finish_access` function ensure that each page that can be modified by a remote node is inaccessible to the local node. The `export_content` function commits the local changes to shared objects and replicates the changes to the remote nodes. The `handle_read_fault` function prepares an object to be accessed by the local node, and the `handle_write_fault` function makes an object ready for a write access. These preparations depend on the consistency model that an object is bound to. In general, they involve registering the object in a service-internal data structure, synchronizing content with other objects and granting the required right for the page. Only the first read and write access to an object within an access sequence is detected. Subsequent read accesses to an object prepared for reading respectively write accesses to an object prepared for writing are invisible to the storage service.

In Figure 2.10, the call graph for hybrid access control is shown. Page faults, read and write accesses share the same code for registering accesses. The `write` function can store modifications right away, whereas the modifications to a memory-mapped object can be saved only at the end of an access sequence because of potential subsequent writes.

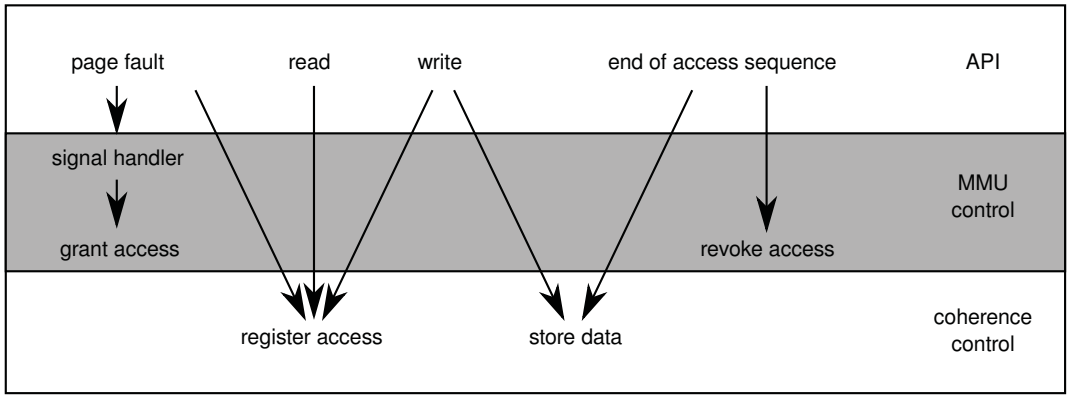


Figure 2.10: Call graph for hybrid access control

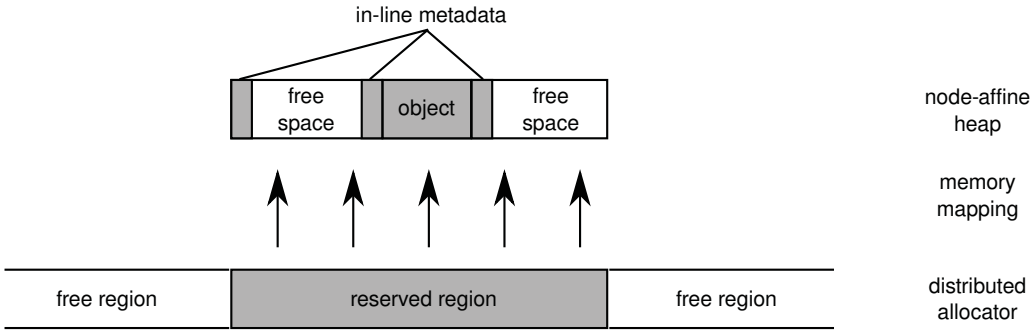


Figure 2.11: A heap created by memory-mapping a distributed region

2.3.3 Stacked allocators for small objects

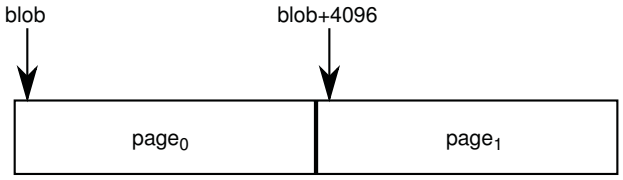
In some cases, developers of distributed application wish to put programming-language objects in the shared storage. Programming-language objects are often small, in the order of magnitude of 10 or 100 bytes. Typically, many small objects are created and deleted over the runtime of an application. For example, online games often share a whole distributed scene graph, in which each node is represented by a shared object [177]. Distributed applications often feature good locality of reference between small objects, because computing nodes mostly store references to objects created by themselves beforehand.

The region-based object management described in Section 2.1 applies to objects of any size. However, frequent creation and deletion of small objects will lead to a high runtime overhead for region management. Additionally, the storage overhead caused by metadata is relatively high for small objects. Therefore, a distributed data store should take advantage of typical object characteristics, exploit locality of reference and aggregate metadata for similar small objects.

The basic approach to handling small objects is the usage of stacked allocators [90]. A low-level allocator provisions nodes with large regions of distributed storage, whereas a high-level node-affine allocator creates small objects inside the regions. The stacking can be done recursively with more than two layers of allocators.

If the low-level allocator provides memory-mapped objects, it is possible to manage small objects with a legacy dynamic memory allocator such as dlmalloc [124], Hoard [22] or StreamFlow [174]. A dynamic memory allocator gets hold of a large chunk of memory and uses it as a heap. Figure 2.11 shows how to create a distributed heap by memory-mapping a distributed region.

With many small objects, an allocator's performance depends on efficient use of locality of reference. An allocator can increase locality by storing metadata *in-line*, that is, directly in the heap, using techniques such as boundary tags or free lists. The consistency of in-line meta-data is handled by the




```
void first()                    void second()
{                               {
  if (0 == blob[0])               if (0 == blob[4096])
    blob[4096] = 255;               blob[0] = 255;
}                               }
```

Figure 2.12: Concurrent access to different parts of a BLOB

storage consistency protocol (see Chapter 3). As a side effect, in-line metadata improves the allocator's portability, because few state resides outside the heap. Another means to enhance locality is to use one heap per thread of execution. Per-thread heaps avoid contention and the need for synchronization when searching for free storage. The dlmalloc allocator provides the *mspace* concept for thread-local allocation.

Using in-line metadata for per-thread heaps relieves the remote free problem. The storage consistency protocol takes care of returning freed objects to the respective heap, regardless of the heap being local or remote. For example, the dlmalloc allocator internally stores a footer encoding the respective mspace with each object.

The allocation of small objects on memory pages gives rise to a phenomenon known as *false sharing*: Collocation of multiple objects on one page makes it difficult to attribute accesses to individual objects. Chapter 5 addresses false sharing in depth.

2.3.4 Large objects

In the context of database systems, a *binary large objects (BLOB)* is an unstructured large chunk of binary data. The information a BLOB holds, for example an image or executable code, is not interpreted by the storage system. A memory-mapped BLOB's size can exceed the hardware page size, such that an in-memory store may need to aggregate memory pages. To enable transparent accessibility, a memory-mapped BLOB must consist of adjacent pages.

Allocation of adjacent memory pages is simple using the region-based approach from the Subsection 2.1.4. However, aggregated objects have different access semantics than objects on a single page. With aggregated objects, multiple nodes can concurrently modify different pages belonging to the same object. In the example shown in Figure 2.12, two nodes can check the if-conditions and execute the following statements concurrently without violating strict consistency. On the one hand, the option to change different parts of an object independently increases parallelism. On the other hand, concurrent modification of the same object can destroy the consistency of the object's internal structure.

The storage system can prevent the counter-intuitive concurrent modification of BLOBs by faking read accesses to each page of the BLOB. If the whole BLOB is treated as being read, concurrent write accesses cause read-write conflicts. Faking write accesses would incur unnecessary overhead, because the BLOB would seem to have been updated, increasing the read-write conflict rate and resulting in expensive transfer of bogus data updates. Concurrent read accesses do not cause any conflicts and are not an issue. The exact semantics of access conflicts are defined in Chapter 3.

2.3.5 Built-in nameservice

Distributed nodes that wish to collaborate over a storage service need to agree on the identity of objects. At least they need a means to access a root object that contains references to other distributed objects.

name	signature	semantics
`nameservice_set`	`object,name`	set the name to reference object
`nameservice_get`	`name→object`	retrieve the object referenced by name

Table 2.4: Abstract nameservice API functions

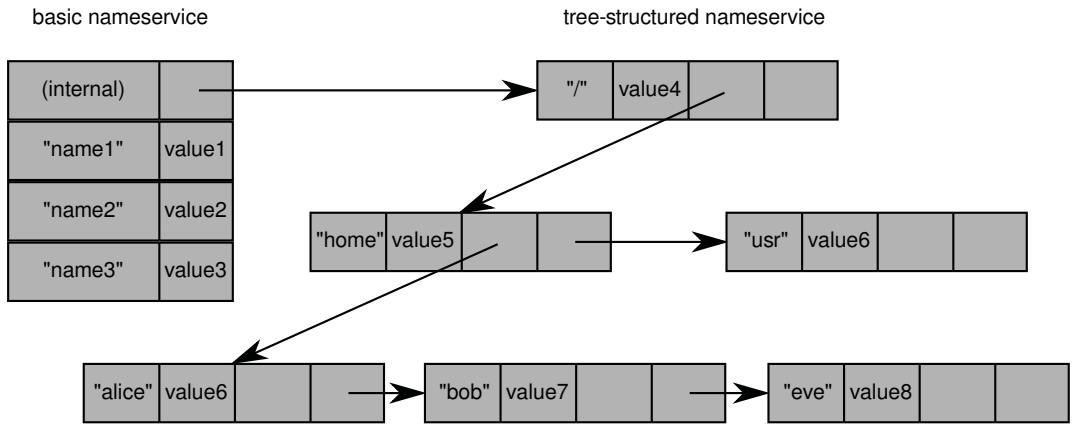


Figure 2.13: A simple nameservice

Application developers can hardcode conventions about the IDs of root objects into their programs, but doing so is not portable and requires the first node to allocate exactly these OIDs. Therefore, developers prefer using a nameservice as a starting point for building object-based structures.

A nameservice is a storage system component that allows to associate application-defined identifiers with OIDs, which the application cannot choose. The nameservice exports functions to associate and retrieve OIDs for supplied names. Table 2.4 summarizes the names, signatures and semantics of the nameservice API functions.

The nameservice component has various ways to implement its functionality. On the one hand, it can provide basic functionality without optimizing for performance, because the application can implement its own functionality on top. On the other hand, a flexible and performant nameservice can relieve many applications from the need to implement their own data organization. Chapter 7 discusses an optimized nameservice. Figure 2.13 sketches the structure of a simple but flexible nameservice. A few entries in a flat table enable applications to anchor their specialized lookup structures. To offer an extensible but non-optimized nameservice, one of the table entries points to the root of a binary tree structure that emulates a tree having a variable branching factor. Each tree node holds the name of the entry as a character string, a pointer to the first child node and a pointer to the next sibling in a single-linked list.

2.4 Related work

The cloud computing era has created new demands on storage systems. Many cloud applications require a storage service that scales well up to terabytes of capacity. Systems such as BigTable [44], GFS [88] and many others provide such services. The availability and scalability guaranteed by cloud storage services are typically high, whereas the consistency models are usually weak. For example, they commonly do not provide atomic multi-key operations. The motivation for offering only weak consis-

tency is that it allows higher scalability [189]. The configurability of these services is usually limited in order to keep interfaces as simple as possible and thereby gain acceptance by application developers. In contrast to typical cloud storage services, this thesis aims at achieving stronger consistency at the expense of reduced scalability. Chapters 3, 4 and 5 extensively address different aspects of designing a strongly consistent storage in a scalable manner.

Object management for distributed storage systems often bases on KBR [150], which uses consistent hashing [110] as a theoretical basis. DHTs such as Chord [178], Pastry [169], CAN [156] and Tapestry [197] build on KBR for scalability and fault tolerance. As a consequence of hashing object names or content to randomly distributed OIDs, DHTs cannot enforce strong consistency and durability, as Knezevic et al. have shown [115]. In contrast to DHTs, this thesis shows that, by choosing OIDs and NIDs in favor of KBR, a storage system can handle object metadata with stronger consistency guarantees and replicate metadata to achieve persistence, and at the same time retain the scalability and fault tolerance properties of KBR. In a similar manner, the storage systems Dynamo [60] and Cassandra [122] partition the object space in a stable manner despite node churn using consistent hashing with an order-preserving hash function. The protocol for mapping storage from neighbors presented in Subsection 2.1.4 resembles Pastry's virtual hosts concept in that it enables distribution of storage load.

The management of distributed objects has much in common with multithreaded dynamic memory allocation. Wilson et al. give an overview and taxonomy of dynamic memory allocation [194]. Many works present implementations for scalable multithreaded allocation [22, 69, 75, 124, 174, 174]. The key idea of these implementations is to exploit locality. Similarly, filesystems are often designed in a locality-aware manner. The first locality-aware filesystem was BSD's FFS, which places inodes of files in same directory placed in same cylinder group, and data blocks of one file preferably allocated in same cylinder [136].

The presented concept of a replicated storage service has much in common with replicated object-based distributed systems such as Orca [19], Thor [129] JuxMem [17] and BlobSeer [144]. Unlike CORBA [188] and DCOM/.NET remoting [137], replication decouples the access to objects and their location. To make an object accessible, replicated systems create local representatives for objects instead of execution operations remotely (see Chapter 4). The alternative concept of moving program code to the data to be processed is currently more popular, but its efficiency depends on benign data access patterns with few data interdependencies (see Chapter 6).

In the recent months, in-memory storage has become the prominent trend for large-scale storage systems. The availability of volatile memory with low access latency at low cost has boosted the demand of upcoming cloud applications. In-memory storage benefits from new technologies to structure storage systems. Early ideas for in-memory storage, called main-memory databases, can be found in the works by DeWitt et al. [67], Li and Hudak [127] as well as and Garcia-Molina and Salem [87].

Many modern in-memory stores do not have a SQL interface, so they are designated as in-memory NoSQL stores. Predominant examples for these systems are RAMcloud [146], GigaSpaces XAP [106], Coherence [52] and Microsoft's AppFabric cache, previously called Velocity [45]. Other in-memory storage systems such as SAP's HANA [76] and Oracle's TimesTen [53] offer a complete SQL interface as well as non-standardized interfaces. This thesis shares the motivation of in-memory NoSQL systems, but in contrast to other systems it focuses on unstructured objects that are allocated and released dynamically.

The hybrid access control mechanism presented in Subsection 2.3.2 includes transparent access detection using MMU hardware. Hardware-based access detection was employed by many distributed shared memories (DSMs). Early implementations such as IVY [127], Mirage [81] and Munin [21] implemented strong consistency models, which causes a high synchronization overhead and is prone to false sharing (see Chapter 5). Later developments, for example TreadMarks [113] and ORCA [19], improved scalability by applying weaker consistency models. This chapter has focussed on object allocation and access detection. Other means to improve storage performance are discussed in the subsequent chapters.

2.5 Summary

Important properties of a distributed storage service such as elasticity, fault tolerance and flexibility are determined by the service's overall design. In order to substantiate the notion of a storage service for the following chapters, this chapter has defined an API for an object-based storage service.

Regarding the scalable management of dynamic objects, it has presented an approach to manage objects using key-based routing for object regions. The concepts used are state-of-the-art, but their combination into scalable dynamic allocation is a new contribution put forward by this thesis. In addition, this chapter has analyzed metadata management with respect to reliable operation. For a storage that associates objects with manager nodes, remote free operations are an important special case, which this chapter has also covered.

A storage service is accepted by programmers only if it is usable with a wide range of application programs. However, the requirements for object size, operations on objects and related properties differ among applications. This chapter has contributed a hybrid access control mechanism, which gives applications the choice between manipulation of objects using API functions and accessing dynamic objects directly as if they were allocated by malloc. For the latter method, the storage service guarantees data consistency using transparent access detection. The implementation of both access methods can share large parts of the code. Further approaches to increase the flexibility of a storage service, which have also been discussed, are support for small objects by stacking allocators, support for aggregating objects to BLOBs and a versatile built-in name service.

3

Flexible transactional consistency

Distributed storage systems replicate objects for performance and fault tolerance. Given that objects on different nodes can be accessed independently, the storage system must ensure the consistency of objects. Consistency models describe the order of data updates guaranteed by the storage system, such that application developers are able to use the storage without experiencing unexpected data inconsistencies [181].

Several consistency models have been defined in the context of distributed shared memory [140]. These definitions differ in the strength of consistency, the possible degree of access parallelism and the tolerance of machine and network failures. The spectrum of possible models ranges from strong consistency to weak consistency. On the one end of the spectrum, *strong consistency* spreads updates eagerly, such that applications always access the most recent state. On the other end of the spectrum, *weak consistency* propagates updates lazily. Weaker consistency often restricts the programming model but benefits the storage system's performance and resilience in terms of increased availability and partition tolerance, as implied by Brewer's CAP theorem (see Chapter 1) [36]. Figure 3.1 summarizes the effect of strong and weak consistency on availability and partition tolerance.

The fundamental challenge in designing a distributed storage system is to find an acceptable compromise between both ends of the spectrum. An optimal storage system matches applications's consistency requirements without complicating the programming model, limiting performance or compromising fault tolerance. This chapter introduces consistency management using transactions on distributed in-memory storage and describes versatile and adaptive ways to implement memory transactions. The underlying concept of distributed transactional memory (DTM) extends software transactional memory (STM) to shared-nothing distributed systems. Performance and scalability of DTM are particularly sensitive to the cost of inter-node communication [31]. Therefore, the subsequent chap-

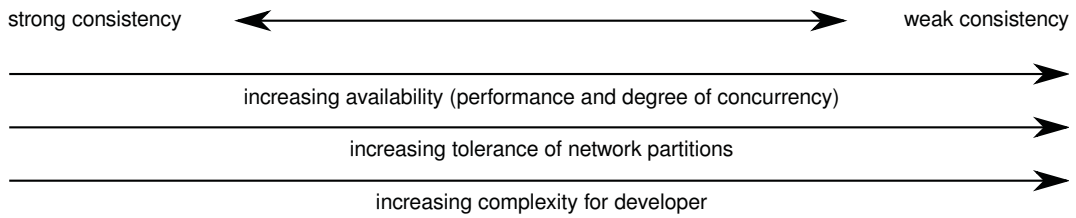


Figure 3.1: Strong consistency versus weak consistency

ters 4 and 5 present techniques to reduce communication in distributed in-memory storage. These chapters build upon the foundations described in this chapter.

This chapter is structured as follows. The first section details design and implementation of speculative transactions on in-memory objects. The second section presents several methods to weaken the consistency of memory transactions in order to improve their scalability in the context of distributed computing.

3.1 Transactional consistency

In a distributed storage system, the communication latency between computing nodes severely impacts performance. Under strong consistency requirements, nodes frequently need to notify their peers about data updates or retrieve updated data from them. An obvious approach to improve performance is to aggregate several operations into a single compound operation. Compound operations allow to transmit operations involving multiple objects in a single network message. For this purpose, database research has defined the transaction concept.

A *database transaction* consists of several read and write operations on a shared storage and adheres to the *ACID* properties [92, 193]. The runtime environment must execute transactions *atomically*, i.e., each transaction's operations must run without being interleaved with operations belonging to different transactions. Each transaction represents a transfer between *consistent* storage states. Transactions must execute in *isolated* manner, without interfering with other transactions. Database transactions are required to be thoroughly *durable*, with changes being written through to disk, such that failures occurring after the end of a transaction do not cause inconsistent state. This section covers the design and implementation of transactions for distributed in-memory storage.

3.1.1 Speculative memory transactions

Transactions on distributed in-memory storage resemble database transactions [31, 119]. The transactional memory concept was put forward by Herlihy and Moss [99]. A *memory transaction* consists of read and write accesses to the shared in-memory storage that take effect atomically, operate on consistent storage and execute in isolation. However, the durability of in-memory storage does not imply data being written to disk. Replicating data to the main memory of other nodes can substitute or complement disk-based durability, because communication over a fast network often outperforms slow accesses to local storage media [146]. Further benefits of replication are discussed in Chapter 4. The section on related work (Section 3.3) reviews work on transactional memory (TM).

The drawback of replication is that it complicates transaction processing. Transactions on replicated data are usually required to be *one-copy serializable*, i.e., the outcome of concurrent transactions must be the same as if the transactions were executed on non-replicated data [26]. The overhead to coordinate replicas and the deadlock rate grow exponentially in the number of replicas [93]. To this end, Gray et al. suggest a two-tier replication scheme that stores single master replicas on reliable base nodes and lazily updated secondary replicas on unreliable nodes [93]. A transactional storage can achieve one-copy serializability using a similar two-tier scheme consisting of single master replicas used for transaction validation and secondary multiversion replicas to improve data availability. Chapter 4 discusses details of integrating replication into a transactional in-memory store.

Optimistic synchronization

The conventional approach of serializing accesses to shared data is locking. Locking requires that the critical code section accessing shared data may be entered only after acquiring a lock protecting the data. The operation that acquires a lock stalls execution as long as another activity possesses the lock. After leaving the critical section, the lock must be released to allow other activities to enter a critical section. The two-phase locking method requires that locks are acquired but not released in the preceding *expanding phase* and released but not acquired in the subsequent *shrinking phase* [192].

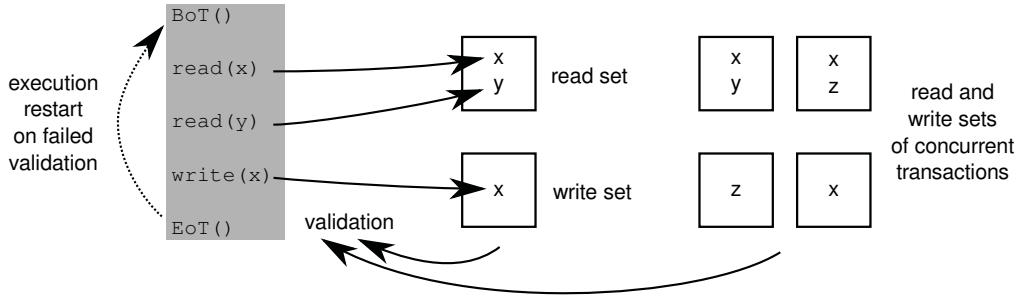


Figure 3.2: Lifecycle of a transaction

Seeing that incorrect locking can cause fatal problems such as deadlocks, the necessity to use locking appropriately complicates application development. In case of low contention on shared objects, locking scales suboptimally in the number of participating processes, because locks are acquired pessimistically, even if they are not needed to ensure serializability [9]. Furthermore, locking in distributed systems can cause distributed deadlocks, which are difficult to detect and to resolve. In contrast to pessimistic locking, optimistic concurrency control [121,193] bases on the idea that, if contention on shared objects is low, global locking of shared data to ensure correct execution is unnecessary in the majority of cases.

However, if concurrency is not controlled using locking, several kinds of conflicts can occur. If the same object is written to by more than one node, at least one update is lost. Violated consistency among several objects causes dirty reads, e.g., reads that return inconsistent data. Undefined interleaving of operations by different nodes leads to non-repeatable reads.

Generally, if two or more concurrent nodes access the same data object, and at least one of the accesses modifies the object, the nodes perceive an inconsistent global state. Therefore, optimistic synchronization must detect misspeculations about global state by validating accesses after speculative execution. Validation needs not determine a serial ordering of accesses, it suffices to decide that the accesses are *serializable*, that is, a serial ordering of accesses exists.

Assuming the absence of conflicts accessing shared data, applications can access objects speculatively and later check whether any conflicts have occurred. The presence of a conflict implies that speculative execution has been invalid and the application must re-execute the respective code fragment.

Speculative execution

Besides bundling several operations, using transactions in a distributed application allows to increase the degree of parallelization, because the runtime system can execute transactions speculatively. For transactional memory, the programmer specifies the beginning and end of speculative execution in the source code using markers such as `begin_of_transaction` (`BoT`) and `end_of_transaction` (`EoT`). Transactional memory implementations in Java alternatively use atomic blocks [118]. In order to simulate speculative execution, the compiler converts the markers into inline code or calls to functions provided by the runtime environment.

Figure 3.2 illustrates the lifecycle of a transaction. At `BoT`, the runtime system initializes the *read set*, which will hold the transaction's speculative read operations, and the *write set*, which will hold the tentative write operations. In the period between `BoT` and `EoT`, the runtime records all shared storage accesses issued by the program in the read set and write set. When reaching `EoT`, the runtime validates the transaction's read and write set against the read and write sets of concurrently running transactions to ensure the serializability of the transactions. If the system decides the transactions to be valid, it commits the speculative modifications to the shared storage. However, if it finds the transactions not serializable, it determines a subset of transactions obstructing serialization, discards their speculative changes and restarts them from `BoT`. Subsection 3.1.4 details the implementation of transparent application restart for speculative execution.

Nested transactions

In order to improve the versatility of transactions, a number of concepts has been developed to define how a single node can run multiple transactions at the same time. A *nested transaction* is a transaction that executes entirely inside another transaction [163, 192]. Compared to non-nested *flat transactions*, nested transactions increase the complexity of implementation, because they require recursive speculative execution.

The usability of a transactional in-memory store benefits from nested transactions. For example, library functions can use transactions to access shared objects, regardless of whether the application calls them from inside or outside of a transaction.

In the *closed nested transactions* model, inner transactions run in isolation until the outermost transaction commits [143]. The concept of flattening closed nested transactions simplifies validation and restart [139]. Flattening ascribes all accesses within nested transactions to the outermost transaction. An implementation of flat nested transactions counts the nesting depth for each node. Only `BoT` and `EoT` operations at the outermost nesting level take effect, whereas nested operations have no other impact than incrementing or decrementing the nesting depth.

3.1.2 Validation

The validation procedure decides whether a transaction is consistent with concurrent transactions. Validation compares the read sets and write sets of concurrent transactions. According to the consistency property of memory transactions, a transaction accessing or producing inconsistent state is detected to be invalid. Inconsistent state is caused by concurrent transactions that conflict with respect to the same object. A *write-write conflict* exists between two concurrent transactions that both write the same object. A *read-write conflict* exists between concurrent transactions where one transaction reads a value that is overwritten by an earlier concurrent transaction, or where one transaction reads a value that is produced by a later concurrent transaction. Concurrent read accesses by several transactions do not harm correctness, because they do not modify data and therefore do not impact speculative execution. In contrast, execution is non-serializable in case of a *dirty read*, where a transaction sees uncommitted changes of another transaction, or a *premature write*, where a transaction overwrites uncommitted changes of another transaction.

Validation strategies

The comparison of transaction's read and write sets can be accomplished in several ways. The validation strategies differ in the set of transactions to compare and in the information required to decide about validity of transactions.

Forward validation compares the transaction to validate against all concurrent still active transactions [129]. The validating transaction cannot have read objects modified by still active transactions, because these modifications have not been committed yet. The validation process needs to ensure that the concurrent active transactions do not read any objects that the validating transaction writes to. Otherwise, the active transactions would have read outdated values. In the example of Figure 3.3, forward validation neither compares the validating transaction 1 with transaction 2, which is not a concurrent transaction, nor with transaction 3, which is concurrent but no longer active. If forward validation finds the set of transactions under examination to be incompatible, none of the transactions has been committed yet, such that the algorithm can choose which transaction to abort. This choice has the benefit that it allows to implement fairness strategies, such as reordering of transactions to prevent conflicts or choosing a short transaction to abort [14].

Backward validation, the complement to forward validation, compares the validating transaction with all concurrent and already committed transactions. The already committed transactions cannot have read objects modified by the validating transaction, because these modifications have not been committed yet. The validating transaction must not read any objects invalidated by committed transactions. Otherwise, the validating transaction would have read outdated values. In case of backward

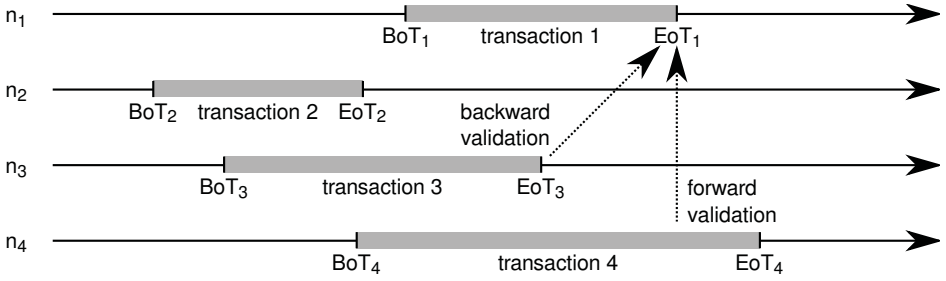


Figure 3.3: Validation strategies

validation, the set of transactions under examination contains only one transaction that has not yet been committed, such that there is no choice about which transaction to abort. Considering that the transaction to abort is always the transaction currently under examination, backward validation does not require communication with other nodes in case an invalid transaction is detected. In the example of Figure 3.3, backward validation neither compares the validating transaction 1 with transaction 2 nor with transaction 4, which is concurrent but still active.

An alternative strategy to backward validation and forward validation is *timestamp ordering* of transactions [6]. A global ordering of transactions can be based on loosely synchronized clocks. On the one hand, timestamp ordering avoids multiversion handling, because all nodes agree on a current global time, which implies the existence of a single current version. On the other hand, the predefined ordering can cause transactions to abort even if they are serializable.

Validation and commit of a single transaction must execute atomically, because a definitive decision about validity can be made only while the storage remains unmodified for a short period of time. However, validation can already yield a negative result while a transaction is still running. Pre-validation checks whether accesses of the local transaction are inconsistent with concurrent transactions that have committed already. In these cases, the local transaction cannot commit, such that local pre-validation helps avoid the overhead of distributed validation [125]. If pre-validation has detected a conflict, it can immediately restart a failed transaction. However, immediate restart can result in a deadlock if the restarted code is holding a local lock that is not automatically released by speculative execution. Ensuring that no locks are held can be difficult. For example, the standard library uses locks to serialize input or output, and these locks are unknown to both the transactional storage and the application code. At the end of the transaction, when all its accesses are known, restarting the transaction is safe, if lock and unlock operations are correctly paired within the transaction.

Multiversion objects

Allowing different versions of objects to coexist is a powerful approach to increase the parallelism in a distributed system [132]. Multiversion concurrency control has first been described in David P. Reed's dissertation [157]. The overhead of storing multiple versions per object can be mitigated by deleting unused versions. Subsection 4.2.3 presents a technique to delete obsolete replicas in a distributed storage by taking advantage of distributed information about transactions.

Multiversion concurrency control of distributed transactional storage has several advantages compared to the assumption of a single version that is always consistent. The possibility to access outdated object versions guarantees that a transaction can always access a consistent snapshot, even though the snapshot might be outdated by concurrent write operations. Therefore, transactions on multiversion objects can execute safely until they reach `EoT` without causing dirty reads or premature writes. A transaction is never required to abort immediately, because inconsistencies among the versions of different objects cannot occur. Figure 3.4 gives an example for an inconsistent snapshot in a single-version transactional storage. The intended invariant is that the sum of x and y shall always equal zero. However, transaction 1 perceives an inconsistency between x and y, such that it cannot continue executing safely after reading y. A multiversion storage would simple present transaction 1 the previous version

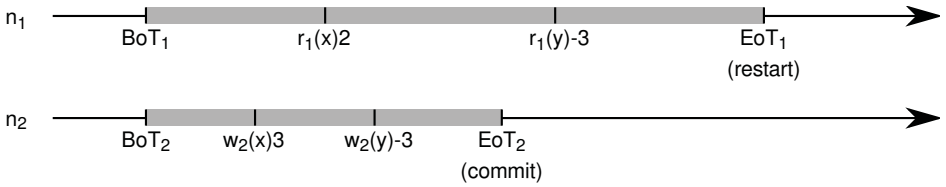


Figure 3.4: An inconsistent snapshot in a single-version transactional storage

of y and restart the transaction when it reaches EoT.

Besides guaranteeing the availability of consistent snapshots, multiversion objects enable further optimizations. Transactions that only read objects belonging to a consistent snapshot need not be validated at all. If applications do not care about read-write conflicts, snapshot isolation provides a weak form of transactional consistency that is based on consistent snapshots [165]. Subsection 3.2.2 discusses snapshot isolation in detail.

The techniques used to generate and compare object versions have great impact on the performance of multiversion object management. During object access and during transaction validation, the distributed storage must frequently decide whether an object version is compatible with a transaction. The validation subsystem must avoid any network communication for these decisions.

To enable efficient comparison between object versions and transactions, the subsystem can associate transactions with transaction identifiers (TIDs) and set the versions of objects modified in a transaction to the respective transaction's TID. A simple solution to the problem of comparing versions is to assign monotonically increasing TIDs, which can be compared locally using simple integer arithmetic. The implementation of monotonically increasing TIDs requires that TIDs are determined either using distributed group communication or using a central component. This restriction is outweighed by the efficiency of comparison, considering that version comparisons are much more frequent than the creation of new versions. A globally accessible timer, which could be used as a version counter, does not exist in a distributed environment. Any other solution for version comparison requires translation tables and network communication to establish a relationship between versions and transactions.

The width of TIDs restricts their numerical range. Monotonically increasing TIDs can eventually overflow. A simple but practical solution to the wraparound problem is to define two alternating phases in the runtime of the storage service. During generation of new TIDs, the TID counter is allowed to transition between the phases only if all nodes are working in the same phase. Given that the wraparound happens rarely compared to the creation of new versions, it suffices to disseminate the information about which node has reached the next phase slowly using an epidemic protocol [61] The information can be piggybacked on other network messages, such that it does not cause extra communication.

Transaction dissemination

Depending on the validation strategy used, nodes participating in the distributed storage need to inform each other about write sets and read sets of tentative or committed transactions. Besides version management, the strategy of communicating read and write sets among nodes is an important design factor for transactional storage. The high communication latencies in a distributed storage system require taking special care of scalable communication patterns.

In general, participating nodes should know in advance which versions to access, such that accesses to old versions and consequent transaction restarts are rare. In a practical implementation, a special wildcard value to request the latest known version simplifies accessing an object that a node has never encountered before. However, notifying all nodes of all objects speculatively accessed or updated by remote transactions causes a high network load.

For transactions on distributed storage, backward validation is a reasonable strategy, because it requires only sending write sets to remote nodes. Forward validation would require transferring tentative read sets, which are usually much larger than write sets. Besides, the potential fairness benefits

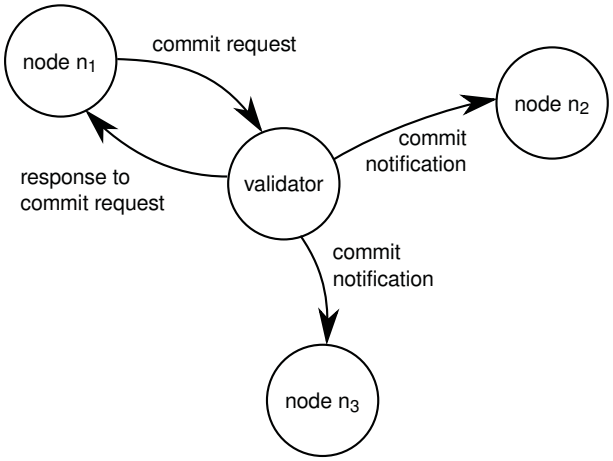


Figure 3.5: Centralized validation in a multiversion DTM

of forward validation are hard to enforce in distributed settings. However, nodes can conduct forward pre-validation locally to detect invalid transactions before starting distributed validation.

The validation procedure can check the serializability of transactions using either a distributed peer-to-peer or a centralized server approach. Distributed checking of serializability is relatively complicated. First, the storage must ensure that at most one peer at a time is allowed to validate. Often a circulating token serves as an arbiter, but issues such as token duplication or token loss complicate practical implementation [142]. Second, in order to have a global view of the distributed versions, the current validating peer must know all previous transactions. Third, peers with slow network connection or weak processing power obstruct the system.

The centralized approach avoids many of the drawbacks of distributed serializability checking. Ensuring single-threaded validation is trivial with a single server. Only the validating node needs to have a complete view of the distributed versions, whereas the other nodes can cache information about versions to improve accessing the right version and to pre-validate transactions. The system can elect a powerful and well-connected node as validating node. However, the scalability of the centralized approach is limited if the server is under heavy load. A backup server or a checkpoint-restart mechanism can make the centralized approach fault-tolerant.

Figure 3.5 shows an implementation of centralized validation in a multiversion DTM. To validate its transaction, the node n_1 sends a commit request to the validator node. The commit request contains the transaction's read and write sets including the respective versions of the objects. The validator checks that the transaction has read or overwritten objects in their current version. If validation succeeds, the validator assigns the transaction a version identifier, sends the requesting node n_1 a positive reply including the transaction's version and notifies the other peers n_2 and n_3 of the newly generated versions by sending them the transaction's write set. The notification enables the peers to pre-validate their local transactions. If validation fails, the validator sends the requesting node n_1 a negative reply.

Transaction history

The fundamental operation during validation is the comparison of accessed object's versions with the committed versions. Accessed object's versions are contained in the validating transaction's read and write sets. The committed versions are stored in the write sets of committed transactions. Implementations of TM must keep these write sets in a *transaction history*, which has a queue structure where new transactions are prepended to the head.

Based on the transaction history, a TM can implement fault tolerance with respect to transaction validation. A node missing a write set can ask his peers to retransmit the commit notification. Failures during validation are especially sensitive, because the transaction must appear to take place atomi-

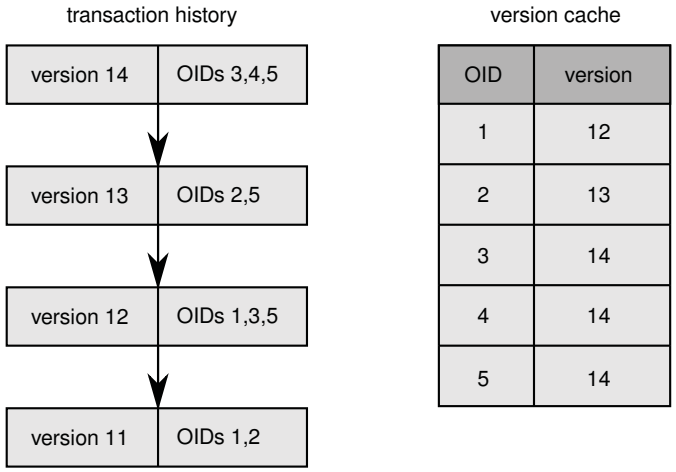


Figure 3.6: Transaction history and object version cache

cally. Two-phase commit protocols have difficulties as soon as any participating node terminates unexpectedly, because each commit must be acknowledged by all nodes. Similarly, token-based commit protocols are prone to failures of arbitrary nodes. If a failing node holds the token or is acquiring or releasing the token, a difficult token recovery process must take place in order to avoid token duplication. In contrast, central validation and commit are impacted only by failures of the validating node. Failures of other nodes can be masked using transaction history information. A node acquires the history information either from its peers or from the central validator.

To determine the committed version for an object, the validation procedure can traverse the queue of committed transactions, starting with the most current transaction, until it finds the object. Validation is a frequent and performance-critical operation, making it a hot spot for optimization. However, the naïve approach to determine an object's version is inefficient in both time and space requirements. An access to an object that has last been modified long ago causes a long traversal through the transaction queue. Furthermore, the approach requires keeping all write sets of committed transactions. As an improvement, the latest versions of each object can be cached in a hashtable. Caching object versions allows retrieving an object's current version in constant time, and only requires a storage size linear in the number of objects. Figure 3.6 illustrates the combination of a transaction history with an object version cache. An alternative data structure to store transaction histories is for example a Bloom filter [29], which allows to aggregate information about many objects modified by a range of transactions in constant time. A similar approach has been used by Chang et al. to summarize whether a particular entry is present in a file stored in the BigTable storage system [44].

In order to avoid unlimited growth of the transaction history, the transaction management must determine the oldest version potentially needed for validation. From the responses to commit requests, the validating node can infer the version of the distributed storage each node has last seen. It can thus calculate the minimum storage version that is still needed and notify its peers either periodically or by piggy-backing the information on other messages. Lazy dissemination usually suffices to maintain the history, but in case of memory shortage, nodes must retrieve the information actively. The subsequent chapter discusses the related aspect of how to delete replicas that are no longer needed in Subsection 4.2.3.

3.1.3 Local commits

Network communication limits the performance of a distributed transactional in-memory store. Thus, a means to improve performance is to avoid communication among the nodes. Checking serializability generally requires communication with the other nodes of the distributed system to retrieve their

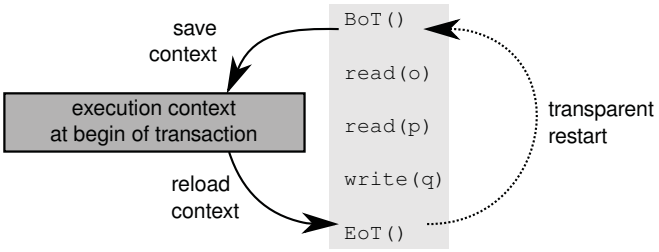


Figure 3.7: Transparent restart of transaction execution

transaction's read and write sets. However, there are cases where a node can commit without contacting other nodes at all.

If a transaction writes only objects not replicated elsewhere, the runtime system can execute a *local commit* for that transaction, meaning that the object state is upgraded without further validation against other node's transactions. Local commits do not compromise correctness of transaction processing, because non-replicated object cannot cause data dependencies at nodes other than the validating node, thus they do not affect serializability. In order to find out whether a local commit is permissible, the validation procedure must disable replication while checking the replication state of the written objects.

Read-only transactions are transactions that do not write any object. On a multiversion storage that provides snapshot isolation as discussed in Subsection 3.1.2, read-only transactions are a special case of local commits. Given that the write set of a read-only transactions is empty, the runtime system needs not validate read-only transactions at all.

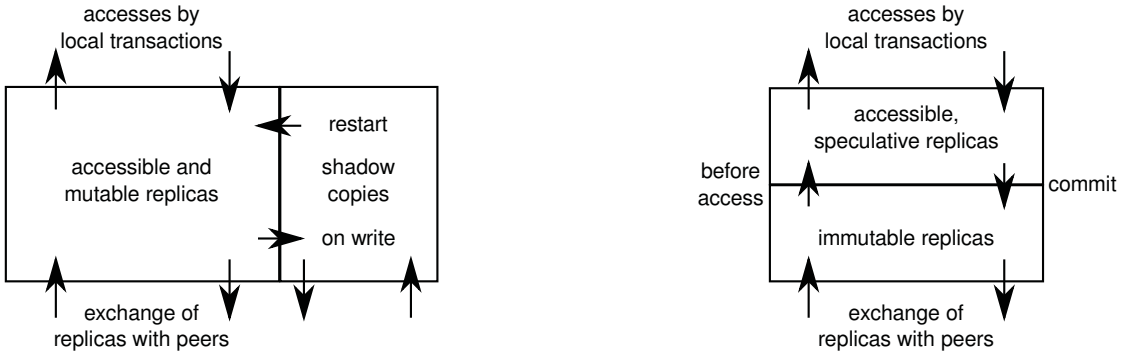
Although non-replicated objects make local commits possible, the lack of replicas interferes with fault-tolerance [88]. In case a node holding non-replicated objects fails, the storage loses information. Therefore, it is useful to distinguish between normal replicas, which are used to reduce access latency, and backup replicas, which serve only to restore system state in case of node failures. The backup replicas must not be accessed like normal replicas, because they are not considered during validation of transaction. A committing node checks for each object in the write set whether the replicas matches the requirements on fault-tolerance. The requirements are that there exists a certain number of replicas, for which the application can defined the minimum, and that the replicas are distributed in a certain manner. If any condition is not fulfilled, the storage creates further backup replicas at nodes specified in object's metadata. These nodes should be either logically related or physically close, such that they can be retrieved fast after a failure. Distributing replicas to different racks or data centers improves the fault tolerance further, because it makes the storage more resilient against power outages [122].

3.1.4 Transparent speculative execution

Without support from the runtime system, speculative execution of memory transactions complicates application development. The application must record all accesses to shared storage and repeat the transaction in a loop until validation succeeds. To provide a convenient programming interface for speculative transactions, the system must implement transparent execution of transactions. In detail, speculative execution requires saving CPU and memory state at the beginning of a transaction, verifying and granting accesses during transaction execution, committing the CPU and memory state after end of a successful transaction, and conditionally reverting the state in case of access conflicts.

Saving and restarting execution context

Figure 3.7 outlines the transparent saving and restarting of a transaction's execution context. Saving the execution context at the beginning of a transaction implies copying the CPU register to memory. Which registers need to be saved depends on the specific CPU architecture. If saving and restoring the CPU state is implemented as programming language functions with their own stack frames, only those



(a) shadow copy based rollback

(b) rollback using immutable replicas

Figure 3.8: Storage rollback

registers that are preserved across function calls need to be saved, because the remaining registers are restored on returning from the function call anyways. For example, according to the AMD64 ABI definition [134], the transactional storage needs to save only the registers `rbx`, `rsp`, `rbp` and `r12` through `r16`, as well as the floating point, MMX or SSE registers if they are used in the transaction. To restore the CPU state, the transactional storage loads the saved register content from memory. Finally, it jumps unconditionally to the restart label in the `BoT` function.

In addition to the CPU state, a program's execution context comprises the function call stack. The AMD64 ABI defines that the stack grows in inverse direction towards lower memory addresses [134]. A special CPU register indicates the top of the stack, but the transactional storage needs to find out the bottom of the stack manually. Unless the compiler has used special optimizations, each stack frame of a called function contains a pointer to the stack frame of the calling function. Thus, the transactional storage can find the bottom of the stack by following the chain of stack frame pointers. To this end, the GCC compiler provides a special function called `__builtin_frame_address`. If the stack does not contain frame pointers, the transactional storage needs to determine the bottom of the stack by asking the operating system or by assuming a certain convention for stack layout. Given that the bottom of the stack remains constant during program execution, determining the bottom of the stack during initialization helps avoid the overhead of traversing the frame pointers during each transaction and circumvents missing frame information due to compiler optimization.

Rollback of distributed storage

Transactional accesses to distributed storage are speculative. When rolling back the transaction, the storage system must revert the effect of speculative modifications. In case a concurrent transaction updates an object, speculative read accesses may return modified values after restart.

The isolation property of transactions requires storage modifications to become visible only at the end of a transaction. If speculative modifications of a failing transaction have already spread to concurrent transactions, dirty reads and premature writes require the transactional storage to cascadingly abort dependent transactions [94]. To avoid the intricate handling of cascading aborts, a storage system that allows modification of accessible replicas must create shadow copies on speculative write accesses, as shown in Figure 3.8a. On a write access, the system creates a shadow copy from the original content of the object, such that it can revert the object when restarting the transaction. If a peer requests a replica of an object that is currently being modified by a local transaction, the storage system must reply with the shadow copy in order not to distribute speculative state. When the transaction finally terminates, the storage system deletes the shadow copies, which are no longer needed.

In contrast, a storage system that stores immutable replicas separately from accessible replicas needs not create shadow copies, because an object can always be restored from an immutable replica.

	outside transactions	within transactions
creation/deletion of transactional objects	runtime system needs to fake transactional semantics	automatically restarted
creation/deletion of non-transactional objects	not impacted	runtime system needs to support restartability

Table 3.1: Handling of object creation and deletion

Besides simplifying object access, the separation of immutable and accessible replicas enable the storage system to store newly created immutable object versions in parallel to applications working on a consistent view of accessible replicas.

Figure3.8b illustrates the advantage of immutable replicas. The storage implementing immutable replicas creates accessible replicas upon object accesses. To avoid reloading accessible replicas that are unmodified, read operations store the version of the respective replica. Write operations store an undefined version to force reloading the replica after a restart or in a later transaction.

The overhead of an additional object copy for read operations is relatively small, considering that the copy needs not be reloaded until a new object version arrives or the object becomes modified. Furthermore, the separation of mutable replicas from immutable replicas integrates well with multiversion replication, which will be discussed in Section 4.2.2.

Restartability of local memory and system resources

Depending on the convention about transaction execution, the rollback mechanism also applies to local memory such as the program's heap, global variables and memory mappings. Modifications of local memory can be tracked using MMU-based access detection (see Subsection 2.3.1). However, detecting all memory modifications incurs a high overhead, because each accessed memory page causes a page fault, and write-protecting the entire virtual address space at the beginning of a transaction takes considerable time. Therefore, it is more practical to rollback only the shared storage and rely on the application developer to anticipate repeated execution when accessing local memory. For example, the programmer should use non-reentrant functions with care, because they access local memory [142].

Programs access operating system resources such as files and network by means of system calls. Using library interposition [91], the transactional storage can intercept system calls. However, the wrapper functions need to record and mimic the behavior of the system calls most precisely, and still the interposition has limited transparency [80].

An alternative to restartable system resources are compensating operations [190]. As detailed in Subsection 3.1.2, the use of locks in combination with immediate restart can cause deadlocks. If the runtime system can detect the acquisition of locks, it can implement a compensating operation that releases the locks before restarting the transaction.

Restartable object allocations

The creation and deletion of dynamic objects can occur outside of transactions as well as within transactions. Therefore, the restartability of dynamic objects requires special handling.

Table 3.1 differentiates between handling of transactional and non-transactional objects outside of transactions and within transactions. In this consideration, transactional objects are defined using in-line metadata (see Subsection 2.3.3), whereas non-transactional objects are defined using external metadata that is not subject to transparent restart. Examples for non-transactional objects are objects in the non-distributed heap and large object regions as described in Section 2.1.

Allocation or freeing of transactional objects requires transaction context to access consistent in-line metadata. To establish a transaction context, the runtime system can automatically wrap these operations in transactions. Within transactions, creating and deleting transactional objects is subject to transparent restart of in-line metadata.

The runtime system must support allocation and freeing of non-transactional objects by saving and conditionally restoring allocation state. To be able to revert allocations on transaction restart, the runtime must register all creations of non-transactional objects within transactions. Furthermore, non-transactional objects can be deleted only at the end of a transaction, because they must be retained in case the transaction restarts.

3.2 Weak consistency within transactions

Transactional consistency implements a speculative but rather strong consistency model. Serializability of transactions assures that a total order of transactions exists. As explained in the Subsection 3.1.2, guaranteeing serializability requires a considerable amount of communication. With the central validator approach, the amount of messages sent is linear in the number of nodes. In an environment with high communication latencies, many nodes and frequent commits, token-based validation requires even more messages to be sent [142]. Two-phase commit and derived protocols have a similar communication overhead [23].

In practice, transactional consistency is stronger than necessary for some well-parallelized computations. For example, computations that adhere to the MapReduce programming model proceed in parallelized phases, such that data updates occur only at the end of each phase. Chapter 6 describes conventional and extended MapReduce models in detail.

The way how an application uses storage has a large impact on storage performance. In case of transactional memory, short transactions allow a higher degree of parallelism, because they keep read sets and write sets small and thus cause fewer conflicts. Approaches for a transactional storage service to support short transactions are discussed in the section on related work (Section 3.3).

To avoid unnecessary synchronization and improve scalability, a programmer should take advantage of weakly consistent operations where possible [40]. These operations must be implemented by storage service, and their weak semantics should not impact correct execution of applications. This section discusses several options to exploit weak consistency within transactions. The first subsection analyzes the interaction of transactional statements with program code outside transactions. The second subsection discusses several means to give applications control over the validation process, and the third subsection examines the coexistence of transactional consistency with weaker consistency models.

3.2.1 Weak atomicity

Programs need not execute all of their code in transactions. In contrast, program sections that do not contend on shared data can execute in parallel outside of transactions. Code outside of transactions can potentially accesses shared data, a fact which gives reason to the definition of weak atomicity and strong atomicity [133].

Strong atomicity requests that transactions execute atomically with respect to other transactions and code outside of transactions. Therefore, each instruction outside of a transaction appears to be a transaction on its own. *Weak atomicity* only requires transactions to execute atomically with respect to other transactions and leaves the semantics of accesses outside of transactions open. With weak atomicity, transactions are serializable only against other transactions, such that code outside of transactions is not synchronized with respect to transactions.

Weak atomicity requires less serialization that strong atomicity, such that it is better suited for a distributed system. Considering that an application developer can always enforce atomicity by writing code in a transaction, weak atomicity does not restrict the programming model.

Relaxation	Impact
Non-validated accesses	Reduce size of read set and write set
Snapshot isolation	Decouple read validation from write validation
Application control over validation and restart	Ignore conflicts
Merging transactions	Optimize validation

Table 3.2: Relaxations to weaken transactional semantics

In cases where weak atomicity is insufficient, the storage can emulate strong atomicity by executing individual operations in transaction context. For example, to guarantee the consistency of shared storage for dynamically allocated objects, the storage must wrap object creations and deletions in transactions, as explained in Subsection 3.1.4. Using flat nested transactions (see Subsection 3.1.1), the nesting of transactions only induces a small runtime overhead but requires only the minimum of distributed serialization [40].

3.2.2 Opaque validation

The degree of parallelism that is possible among concurrent transactions depends on the contention on shared data. Only transactions that do not conflict are allowed to commit in parallel. An effective means to increase parallelism is to reduce the number of access conflicts. Ignoring conflicts in the validation procedure compromises the consistency of shared storage. Therefore, the application code, which accesses shared objects, needs to specify explicitly where checking for conflicts is disposable. Giving applications partial control about the validation process leads to *opaque validation*. Table 3.2 summarizes several approaches to reduce conflicts by means of opaque validation. These approaches are discussed below.

Non-validated accesses

Some algorithms allow the programmer to specify in advance which data is prone to access conflicts and which data is not. For such cases, a transactional memory can offer an interface for accesses that are not added to the transaction's read or write set. Alternatively, an interface to remove objects from the read or write set provides equivalent functionality. If the programmer of a transactional application can preclude conflicts on certain variables, he can use the interface to hide accesses from validation. Abbadi, Harris and Moore propose the dynamic separation model that allows to switch between transactional and non-transactional accesses during program runtime [3].

Which interface a programmer can use to exclude accesses from validation differs between implicit and explicit transactions. *Implicit transactions* register all accesses transparently with respect to the application, whereas *explicit transactions* require the application to access objects through accessor and mutator functions [40]. In case of implicit transactions, the programmer must remove objects from the read or write set after accessing them. In case of explicit transactions, the storage interface can allow controlling access registration. On the one hand, dynamic control about accesses being transactional or non-transactional enables certain programs to achieve better parallelism. On the other hand, any interface to mark non-validated accesses complicates the programming of transactional applications.

Snapshot isolation

Applications can sometimes agree to read slightly outdated object content, especially if application logic includes further plausibility checks. *Snapshot isolation* allows this relaxation in that it splits the read and write phase of transactions [165]. However, it does not guarantee serializability of transactions. During speculative execution, multiversion concurrency control provides consistent snapshots for read accesses, as described in Subsection 3.1.2. In the validation phase, snapshot isolation only checks for write-write conflicts. Therefore, writes can appear to happen at an instant younger than the transaction's reads. Applications need to check for relevant read-write inconsistencies themselves, using application-level knowledge about object states. An implementation of Lee's routing algorithm that takes advantage of snapshot isolation is presented in Subsection 6.3.5. Programmers need to understand the semantics of snapshot isolation well in order to implement programs that benefit from it.

Application control over validation and restart

The storage can give applications even more control over the validation and restart mechanism. Application developers can contribute to validation using semantical knowledge about accesses.

In some cases, the programmer can tell in advance that a code fragment will not conflict with a concurrent transaction. For example, code fragments that accesses only local data need not run as transactions, and some applications contain computation phases that access only data that will always be up-to-date. Executing such code in transactions can be useful nonetheless. First, the transaction context retains the benefits of batching data updates. Second, on a multiversion storage, read operations will always access a consistent snapshot. Third, a program can benefit from speculative execution and initiate a voluntary transaction restart on its own, even though the transactional storage does not detect a conflict. Use cases for this feature are algorithms that adhere to the dynamic programming pattern. Dynamic programming allows to skip the computation of partial results, if a better result has already been found. A computation that has already begun can be terminated in an elegant manner using transaction semantics.

The developer can sometimes anticipate circumstances where a transaction is processing a computation whose results are already irrelevant by the time the transaction tries to commit [177]. Application-specific knowledge can allow to identify semantical conflicts before the transactional storage detects access conflicts. Validating or committing a useless result hampers validation of concurrent transactions. In these cases, the application can force the validation to fail without further analysis of accesses.

A transaction usually retries speculative execution until validation succeeds and the transaction can commit. If the results are irrelevant or if the node should continue with another computation, restarting the transaction does not benefit. Instead, the transaction can specify a maximum number of retries. A transaction requesting that it should not be restarted at all resembles Sinfonia's minitransactions [10]. Like the previously discussed approaches for application-controlled validation, non-transparent validation and restart complicate program development.

Merging transactions

To resolve conflicts on simple data structures, the storage system can accept hints how to merge conflicting transactions. For example, when inserting nodes into a graph, a parallel application can specify that `NULL` fields can be overwritten with specific values without causing a conflict. A simple conflict resolution mechanism can effectively help in cases of false conflicts, which are discussed in depth in Chapter 5.

3.2.3 Dynamic consistency

Some applications do not require the same consistency model for their data during the whole runtime. Implementations of algorithm design patterns such as MapReduce and dynamic programming can

contain phases where data accesses are well known in advance. For these applications, the weakened transactional consistency described so far is too strong. A design where consistency management and replication are separated simplifies the implementation of configurable and exchangeable consistency models. This subsection focuses on the coexistence of transactional consistency and weak consistency using synchronization variables as defined by Dubois et al. [72].

The weak consistency model augments each shared variable with a synchronization variable and requires access to synchronization variables to be sequentially consistent. All writes have to be completed everywhere before any operation on a synchronization variables. Accesses are allowed only after operations on synchronization variables have been completed. This weak consistency model is similar to transactional consistency in that accesses to synchronization variables group operations on normal variables.

To use dynamic consistency, the application needs to define which consistency model to apply. The consistency definition can be provided for each object access, per variable or for all variables in a program phase. Specifying the consistency model during each access is very tedious for the application and can cause inconsistencies if the same object is used under different consistency models by several application instances. Per-variable definitions of consistency models offers the same flexibility, but it can require many reconfigurations for phase changes where many objects use a different consistency model. Therefore, binding consistency models to program phases is a practical solution that many typical applications can benefit from. It suffices for the storage service to implement the consistency-related operations and a method to switch the consistency model.

3.3 Related work

Efforts of keeping data consistent in a distributed system are always faced with the performance overhead of distributed communication. General discussions of consistency models and their classification are found in the publication by Fekete and Ramamritham [77] as well as in the literature review by Mosberger [140]. Using relaxed consistency models to improve scalability has been proposed in early publications on distributed shared memory (see Chapter 2) [5, 72]. Many years later, the CAP theorem [89] supplied the theoretical foundation for the interrelationship of the involved concepts.

For scalable storage systems over wide-area networks, diverse consistency models have been suggested. Event ordering in these systems is difficult, because a common time does not exist. A popular approach to order events is to use imprecise clocks, which was suggested for example by Adua et al. [7] and Fox et al. [84]. Eventual consistency, which was proposed by Terry et al. [182], gives only the guarantee that updates will arrive at each replica at some unknown point in the future. Despite this weakness, eventual consistency has become very popular for web services such as those provided by Amazon [189]. Lloyd et al. take the contrary point of view that eventual consistency can expose strange orderings to developers and users, and they suggest a causal consistency model for wide-area networks that can handle conflicts [130].

Transactional memory is a broad research topic that ranges from low-level hardware design to high-level software implementations. Transactional memory that is implemented in a processor is called hardware transactional memory (HTM). On the one hand, the low abstraction level of HTM complicates implementation of unbounded transactions. On the other hand, direct access to processor registers and caches improves the efficiency of implementations. A decade after the first proposal of HTM [99] and several years after the first wave of HTM implementations such as Sun's Rock processor [70], major hardware manufacturers are currently building TM functionality in their processors [51].

The adverse concept to HTM is software transactional memory (STM). It does not require hardware support for running transactions. The implementation in software can take advantage of high-level data structures and more complex algorithms. Furthermore, it simplifies experimenting with new features. STM can adapt flexibly to transactions of different sizes. The idea of STM was proposed by Shavit and Touitou [175] and became popular with implementations such as transactional locking 2 (TL2) [71], TCC [95] and Deuce [118]. The book by Harris [97] gives a broad overview of STM.

Herlihy and Wing distinguish serializability from linearizability, which is a weaker property for

single operations on single objects [100]. In contrast, the transactional consistency discussed in this chapter applies to atomic operations spanning many objects, and the described implementation supports large read sets and write sets. HTMs usually track accesses at the granularity of processor cache lines. STMs have more options for access control, such that some implementations use the MMU for this purpose [2, 48].

Several analyzes have compared the performance overhead and usability by programmers between lock-based synchronization and TM. Saito and Shapiro detail the benefits of optimistic algorithms [171]: Optimistic synchronization improves the availability and flexibility of applications having variable communication patterns in unreliable networks. Furthermore, it enables better scalability through reduced synchronization, and it allows for autonomous and asynchronous collaboration.

Several investigations have compared the usability of TM with traditional lock-based synchronization. These comparisons generally conclude that TM has a positive impact on the programmability of synchronization algorithms, and that the effort of learning to use a new synchronization paradigm is outweighed by the reduction in synchronization errors. Chapter 8 gives a detailed review of these works. In summary, the use of TM seems to increase the productivity of programmers.

Different approaches have been suggested to achieve scalable validation mechanisms. The researchers of the Thor project argue that validation takes place in the best case at one server, because it exploits locality of reference [129] In contrast, the Hyder project uses an efficient distributed validation protocol, in which each node broadcasts its intended read sets and write sets [25]. Hyder validates fast if the intents of all nodes are available. In case of network delays, validation has to block until retarded intent messages arrive. TCC [95] and DiSTM [119] employ similar methods. Ruivo et al. and Couceiro et al. suggest a refined method called partial replication, which allows to restrict the number of recipients of validation messages [55, 170]. Couceiro's PolyCert implements certification based on atomic broadcast and allows the coexistence of different certification protocols. A different approach is to order validations using the timestamp generated by loosely synchronized clocks [166]. In contrast to the mentioned approaches, this chapter has presented a centralized validation mechanism, which accommodates the implementation of the smart replication described in Chapter 4, is simple to implement and fault-tolerant.

Transactional semantics for distributed storage have recently become a research topic. Shrira et al. state early ideas on the scalability of a distributed storage system in their work on the Thor storage service [176]. Their key ideas are split caching and fragment reconstruction. The split caching approach replicates objects among client nodes in a manner similar to a shared-anywhere system, and fragment reconstruction merges changes of object on one page to circumvent false sharing. Müller discusses different commit protocols for wide-area DTM [142]. In contrast to this thesis, his dissertation does not investigate the effects of replication (see Chapter 4) and sharing granularity (see Chapter 5), and it does not detail the applicability of DTM to MapReduce applications (see Chapter 6) or distributed filesystems (see Chapter 7). The APIs of some DTMs differ largely from HTM interfaces. For example, TxCache is a transactional version of the popular Memcached key-value store [151]. Similarly, G-Store provides transactional multi-key access using both reliable and unreliable messaging for keygroups [56]. Sinfonia [10] and Percolator [148] implement special transactional semantics that are tailored towards specific use cases. Contrarily, the transactional storage consistency described in this chapter applies to a broad range of computations on unstructured, variable-size objects.

An important technique to achieve better scalability is the support of multiversioning, which was first proposed by Manassiev et al. [132]. The distributed multiversioning protocol (DMV) they propose acts similar to split caching and fragment reconstruction in that it allows concurrent operations on the same objects. Reordering of transactions allows to minimize the number of failing validations. Like the transparent storage checkpointing and rollback mechanism described in Subsection 3.1.4, nodes on DMV do not update their view of the distributed state immediately after receiving a commit notification, but instead store the new version in a separate buffer until the version is accessed. DecentSTM [27] combines multiversion management with voting and snapshot isolation. Similarly, selective multiversioning benefits long-running read-only transactions with an adaptive garbage collection of potentially accessible objects [149]. Multiversioning is also used to achieve highly concurrent I/O in BlobSeer [144].

A number of techniques have been proposed to reduce conflicts and manage contention in TM. Atoofian et al. suggest a lazy and adaptive validation protocol [18]. In their work on ClusterSTM, Bocchino et al. discuss design choices for large-scale TM [31]. With the object-aware HTM, Khan et al. exploit object structure to reduce false conflicts [114]. The false sharing phenomenon is discussed on Chapter 5 in this thesis. Adaptive concurrency control, which was suggested by Ansari et al., dynamically adjusts the number of transactions executing concurrently with respect to the fluctuating contention [16]. The number of conflicts can also be reduced by transaction reordering [14]. Although transaction reordering was not explicitly discussed in this chapter, it can be used in addition to the other conflict reduction techniques. Carvalho et al. suggest annotations for non-transactional accesses in order to weaken the transactional consistency [41]. Adaptive policies for TM are discussed in Chapters 4 and 5.

Short transactions allow a higher degree of parallelism, because small read sets and write sets cause less conflicts than large ones. However, if the number of conflicts is low anyways, short transactions incur a higher validation overhead, because the application needs more short transactions than long transactions for the same number of accesses. Thus, the application developer must take care of structuring its transactions in a beneficial manner. A TM can additionally support flexible length of transactions, for example using markers for places in the code where validation for the code executed so far in a transaction can take place but is not required, based on the assumption that atomicity beyond the marker is not required. The TM can then validate the transaction asynchronously [142] or it can determine based on the current conflict rate whether a validation is probably useful [31].

In JVSTM [79], read-only transactions can always commit. Write operations are fully optimistic, and the commit phase is mutually exclusive for all threads. Therefore, the storage can use global version numbers. Versioned Boxes hold the history of transactions at each node.

Adya et al. describe a concept that is similar to the transaction history management and shares the problem of truncating old entries in the data structure [6]. Their validation queues store not only committed but also non-committed versions and are used as part of a two-phase commit protocol. The authors suggest using a threshold timestamp and an active transaction record to identify old versions that can be deleted, which can cause transactions to fail if they are too old. The active transaction record is a central component and therefore less scalable than the partly distributed technique proposed in this chapter. The centralized approach for validation allows JVSTM to implement a more precise size adaptation for the history based on knowledge about node's state. to avoid unlimited growth of the data structure [79].

A distributed replicated database is designated as a shared-nothing database [179]. Kemme and Alonso analyze the interdependency of replication and transaction properties for shared-nothing databases in cloud context [26]. They present properties of the Postgres-R replicated database, such as non-distributed transactions, strong isolation and total ordering of transactions with reliable delivery. Independent commit at participating nodes and early lock release reduce the latency of write operations. that are similar to the design of the transactional consistency and especially to the relaxations presented on Section 3.2. Although the presented object-based storage service is not a traditional database, it implements comparable concepts. For example, local transactions correspond to Postgres-R's non-distributed transactions.

3.4 Summary

Handling storage accesses with transactional semantics has a twofold benefit. First, it enables atomic operations on several operations at once. Second, batching operations saves communication bandwidth. Recent research on DTM demonstrates the options for applying STM to distributed storage, where communication latencies are higher than on HTM or shared-memory STM. This chapter has presented the design of a transactional consistency model for distributed storage. More than previous work in this area, the design focuses on integrating several approaches to enhance the flexibility of DTM, among them flat nested transactions, centralized validation, MVCC and efficient transaction history management. This flexibility is achieved using a modular design that focuses on realistic use cases for target applications. A dedicated section has detailed ways to weaken the traditionally strong

transactional semantics. An early, less flexible implementation of distributed transactional memory is described in a publication written by the author of this thesis [141].

4

Smart replication

Consistency protocols define when the results of storage operations becomes visible. The previous chapter has discussed transactional consistency of distributed replicated storage. The considerations assumed that every node can access each object in the system. Data replication duplicates object content on different nodes. A storage can try to make every object instantly accessible at every node, but this effort obviously demands much communication. Besides, the notion of instant accessibility is restricted to a virtual global time, because an exact global time is unavailable in a distributed system.

Replication greatly impacts performance and fault tolerance. In the face of dynamic accesses to shared objects and spontaneous failures, replication must take into account several parameters of the system, such as the update rate, the access probability and the frequency of failures. The complexity of influence factors requires handling replication in a smart way.

This chapter first defines basic notions in the context of replication. Then it describes the architecture of a replication service for in-memory objects that is orthogonal to consistency management. Furthermore, the chapter presents an approach that adaptively invalidates or updates replicas, followed by the characterization of optimizations for the replication service. It concludes with a discussion of related work and a summary.

4.1 Terminology

In a shared-nothing system that partitions objects among nodes, each object is stored at only one node, which is called the manager of the object. A node wanting to read the object asks the manager to transfer the data to him. A node wishing to modify the object sends the requested operation to the manager, who executes the operation on behalf of the node. Considering that read operations and reliable write operations require at least one synchronous round-trip communication, a partitioned storage limits the throughput of operations. In order to support atomic multi-object operations, a partitioned storage must use an expensive coordination protocol such as two-phase commit .

Replication makes objects accessible at several nodes by creating duplicates of the objects. Accesses to local replicas avoid slow communication with remote managers. Besides, replication also benefits fault tolerance, because it allows to reconstruct a replicated object's content if the manager of the object has failed.

To simulate shared objects on a distributed shared-nothing architecture, the storage service must replicate the content of the objects. This section defines the terminology of locality, replication and version management.

4.1.1 Locality of reference

The effectiveness of replication is strongly related to the locality of reference principle, which is an empirical phenomenon of data access caused by program structure and data layout [63,64]. Respecting locality of reference is a means to improve performance of storage operations. Developers can often foster locality of reference when coding data accesses, and storage systems usually optimize object layout to take advantage of strong locality.

Temporal locality

The two aspects spatial locality and temporal locality often occur in combination.

Definition 6 *Temporal locality is present if the same data object is accessed several times in a relatively short time period.*

Applications often reuse or recompute recently used values in the course of their calculation. For example, storing intermediate values for later reuse or iteratively updating a variable cause temporal locality. The notion *often* is obviously imprecise. To observe temporal locality, practical implementations specify the time frame in question, e.g. by using a working set model [62].

Storage systems can exploit temporal locality to reduce the average access latency. For example, caching on a processor is a special form of replication where the processor keeps data that it has used earlier. Processor caches implement simple but time-efficient replacement protocols such as direct or set-associative mappings. Caches at higher levels of abstraction, for example the buffer cache in an operating system can afford more sophisticated but also more time-consuming replacement protocols [199].

Spatial locality

The complement to temporal locality is spatial locality.

Definition 7 *Spatial locality designates the typical case that a program uses nearby data objects together.*

Entities such as scalar variables, programming language objects or files are often created and accessed together. Subsequent elements in an array are often accessed sequentially. Heap allocation strategies cause parts of complex data structures to be placed at nearby memory addresses. Both data structure layout and allocation strategies foster spatial locality.

Spatial locality gives storage systems the opportunity for optimizations. Processor caches operate on cache lines larger than a single machine word, because processors often use adjacent words together. With high probability, data items allocated on stack or heap reside in the same cache line, as well as array elements or characters in a string.

Implications of strong locality

Strong locality enables important performance optimizations for distributed applications. Observations of locality in the past allow a storage to accommodate similar accesses in the future. Optimizations can be implemented in the application or in the storage service. The former approach requires the application developer to make assumptions about the underlying storage architecture. The latter approach needs a storage service that observes and predicts data access patterns.

As detailed above, typical program behavior often causes locality of reference. In addition, application developers can improve locality by implementing algorithms in locality-aware ways. If they know the parameters of the storage hierarchy, such as the cache line size, in advance, they can implement cache-aware algorithms. Cache-oblivious algorithms always perform optimal regardless of underlying storage, but their implementation is intricate [152]. Therefore, applications often foster locality regardless of actual parameters of the underlying storage system.

A storage service can exploit data access patterns to use locality efficiently. Therefore, the storage must adapt to different access patterns with little support from the applications, at best fully transparent for diverse applications. The service must abstract from specific applications and do without a-priori knowledge about data interdependencies. A transparent adaptive mechanism to take advantage of locality of reference needs to monitor data accesses and infer access patterns in order to predict future accesses.

4.1.2 Replication

Before discussing various techniques and use cases for replication, the notion *replication* is defined according to van Renesse and Guerraoui [185] as follows.

Definition 8 *Replication is creating multiple copies of a possibly mutating object (file, file-system, database, and so on) with the objective to provide high availability, high integrity, high performance, or any combination thereof.*

Replication serves the goal of establishing or increasing data availability. In a shared-nothing system, replication enables applications to access partitioned data. A side effect of replicated storage in the context of transactional consistency is that it avoids partitioning among several storage nodes. Given that data is available at each node, the storage system needs not implement distributed transactions, that is, transactions spanning several nodes.

Replication is a key technique to decrements the latency of data accesses, because accesses to nearby replicas usually take less time than accesses to remote objects. It also helps increase fault tolerance, considering that, if a replica becomes inaccessible, the storage can recover the object using backup replicas. The issues that arise in the context of replication have been discussed in Chapter 1.

An application cannot distinguish a replicated copy of an object from the original object. Implementing the replication of read-only objects is straightforward. The storage can create additional copies of the objects as long as space allows to do so. Objects that can be written to are more difficult to replicate, because an update must happen at all replicas at once, such that the outcome of a read operation does not depend on which replica it accesses. Therefore, the handling of updates to replicated objects is of primary interest.

The above definition refers to data replication in contrast to computation replication, a method that executes the same computations at multiple hosts. Computation replication complements data replication, but in practice mostly used to provide fault tolerance. This thesis focuses on data replication.

Research in the areas of databases and distributed systems has defined diverse replication techniques. An important distinction is between active replication and passive replication. *Active replication* means that modifying operations are applied to each replica. Contrarily, *passive replication* means that an update is first applied to one replica, whose content is then copied to the other replicas.

Passive replication can be implemented as either primary-backup or multi-primary replication. With *primary-backup replication*, for each object exists a primary replica, which is the only replica that an application can modify directly. With *multi-primary replication*, applications are able to update any replica. Multi-primary replication allows for distribution of updates and therefore higher scalability, but it requires a protocol to handle concurrent updates [93].

The dissemination of updates can proceed synchronously, which means that the updating process continues execution only after all replicas have been updated. The alternative is asynchronous replication, which allows the updating process to proceed while the updates are still on the way to the replicas. Asynchronous replication enables faster updates than synchronous replication. However, the higher degree of concurrency requires more efforts for ensuring data consistency. Purely asynchronous replication is prone to data loss in cases of failures, because a failing node can hold updates that are not yet replicated on any other node. Therefore, a storage usually creates replicas for reliability synchronously and replicas for improved access latency asynchronously.

A storage service has different opportunities to spread object content during updates. With the write-update scheme, a node that updates an object sends its peers a message containing the updated object version. In contrast to the write-invalidate scheme, the updating node notifies its peers of the

updated object, but it does not transmit the updated content. Consequently, a peer that successively accesses the object must request the replica from the updating node. Write-update pushes replicas to peer nodes, whereas write-invalidate pulls replicas from updating nodes.

The ordering of updates to a specific replicas is described by the concept of *coherence*.

Definition 9 *A storage is coherent if updates to one object appear in the same order for all participants.*

A coherent storage effectively appears to be a non-replicated object, which all nodes access. In contrast to consistency, coherence is concerned only with individual objects. The ordering of operations on different objects is left undefined.

Whenever a participant of a distributed storage updates an object, the replicas of the object transition from old to new content. By refining the state transition and handling versions explicitly, the storage can execute the update procedure with increased concurrency. Multiversion concurrency control (MVCC) treats each state written by an update operation as a new version [24, 157]. To designate a specific version of an object, MVCC uses version identifiers.

Definition 10 *An object version is the well-defined data content for a given combination of object identifier and version identifier.*

The increased concurrency enabled by explicit versioning has several causes. First, MVCC allows to implement replica coherence mostly orthogonal to consistency maintenance. Second, the storage needs not transmit data to define an object's state, which saves network bandwidth and reduces the latency of processing update messages. Third, the version ID can encode additional information as defined by consistency model, for example to validate accesses for optimistic synchronization (see Chapter 3). 3.1.2

4.1.3 Use cases for replication

Replication enables diverse opportunities for optimization. Two widely applied techniques are caching and read-ahead. Replication also benefits fault tolerance. These different techniques have in common that they try to predict the future use of replicas based on analyzing past accesses. The decision whether to replicate also depends on the tradeoff between creating and requesting replicas (see Section 4.3).

Caching

Caching is a simple but often effective replication strategy. Karedla et al. give the following definition for a cache buffer (short: cache) [109]:

Definition 11 *A cache buffer is faster memory used to enhance the performance of a slower memory (a disk drive, for example), known as the backing store. By keeping copies of backing store data, caches can service some requests at faster memory speeds.*

A storage component caches data if it could be accessed soon again. Unless an object is modified since is has been stored in the cache, a request can be satisfied from the cache. Accessing an object stored in the cache is typically an order of magnitude faster than accessing a remote object. Therefore, the benefit of short access latency in beneficial cases usually outweighs the overhead of checking whether an object is available in the cache. Two prominent examples of caching are CPU caches and web caches.

CPUs duplicate recently used main memory sections in caches, hoping that the program will soon access the same or adjacent cells again. Handling data in cache lines of several bytes instead of single byte cache entries enables efficient transfers between RAM and CPU, and it benefits spatial locality. Cache usage analysis must be very efficient, such that cache management does not determine the exact access frequency, but implements simple and efficient evocation strategies.

Caching of static web content greatly improves the access latency. A web browser that reads a page from its local cache or from a nearby website cache (also known as web proxy) can access the content within a few microseconds instead of several hundred milliseconds for remote accesses. Web caching is

limited to static content. Dynamically generated information cannot be cached, because it is generated by the web server based on the state of the web application and on parameters supplied by the client. Websites are becoming increasingly dynamic and interactive, so developers minimize the amount of dynamically loaded content using techniques such as AJAX (asynchronous Javascript and XML). Web caching also has the drawback that updates to static websites disseminate only slowly from the web server over the caches to the clients. Unless a client directs the web cache to reload its data directly from the webserver, the pattern of interaction only allows for eventual consistency.

Read-ahead

Based on observations of object accesses, a storage system can try to predict future accesses. In case of predictable access patterns, prefetching data allows the storage to replicate data asynchronously [187], a technique called read-ahead replication. In beneficial cases, all accesses are satisfied using local replicas, such that there are no remote accesses causing synchronous on-demand replication.

Modern operating systems use read-ahead replication for sequential file accesses. Most operating systems do not interpret structured file content, so they cannot support more complex access patterns. However, an object-based storage system can analyze derive object interdependencies from access patterns. For example, it can correlate the IDs of objects that are accessed during the same time period.

Fault-tolerance

Replication also helps prevent data loss in cases of failures. After the detection of a machine outage, a recovery routine recreates lost object fragments from the backup replicas. The storage can handle network failures similarly, but it must take care of inconsistencies introduced by partitioning. The highly fault-tolerant and distributed Google Filesystem stores three replicas by default [88]. Google Filesystem's multi-level distribution protocol spreads replicas to different racks in a cluster. Thus, it tolerates total failures of whole racks. More detailed descriptions of use cases for replication can be found in a book chapter by van Steen and Pierre [186].

4.2 Replication service orthogonal to consistency

The concepts of replication and consistency are strongly related (see Chapter 3). However, a storage does not need to integrate both concepts in a monolithic implementation. The complementarity of replication and consistency allows for an orthogonal and modular design with separate consistency and replication subsystems.

Compared to a single service handling replication and consistency, an orthogonal design has several benefits. First, orthogonality makes it possible to change specific policies of one module without needing to change the other. The storage developer can implement the replication policy independently from the consistency policy. Different consistency models can use the same replication model. Second, allowing replication to operate mostly independently from consistency allows for a higher degree of parallelism. The decoupling of modules allows replica accesses to proceed independently from consistency checks. Third, the modularity fosters clean interfaces and simplifies debugging. The replication and consistency modules designate object versions using object identifiers and version numbers.

The orthogonal design has some shortcomings. The narrow interface hinders information sharing between consistency model and replication service. Restricted knowledge makes some decisions difficult for a module. For example, the replication module does not know how long it needs to keep a replica before it can delete the replica. If the distributed storage deletes all replicas of a version that some node can still access, a consistent snapshot for that version is not available. A disadvantage of the orthogonal design concerning local commits (see Subsection 3.1.3) is discussed below in Subsection 4.4.1.

name	signature	semantics
`create_replica`	`object,size,source_buffer,node, previous_id,version_id`	create a replica
`get_replica`	`destination_buffer,max_size,object, version_id→destination,version_id`	retrieve a replica

Table 4.1: Basic replication API

4.2.1 Architecture

The proposed replication service implements a narrow version-based interface that is neutral to consistency models. Instead of being tied to a specific consistency model, the service uses version IDs to guarantee a coherent view of the data store. Table 4.1 summarizes the semantics of the two most important interface functions. The function `create_replica` stores an object version in the replica store, and the function `get_replica` retrieves an object version. Consistency models use these functions for local as well as remote operations. If a version is requested that is not available locally, the replication service sends a request to a remote node he assumes to have the version available.

The two replica access functions are furnished with a wildcard mechanism that allows to call these functions even though part of the arguments are still unknown. If a consistency model does not have the payload data for a replica to create available, it can call the function `create_replica` with an empty source buffer. In this case, the replication service stores the metadata in a placeholder for the replica. When the application accesses the corresponding data later on, the metadata helps contacting the node which holds the payload data. Specifying an undefined version to create allows the local commit mechanism to update the payload data for an non-replicated object, as described in Subsection 3.1.3. Similarly, the `get_replica` accepts an undefined object version parameter. It returns the most current version known together with the corresponding version ID. Subsection 4.2.2 details how the replication services determines the most current version. To account for variable object size, the size of the buffer can be specified to the replication subsystem as a function call parameter. If the specified buffer size is an undefined value, it is overwritten by the actual object size, and if the object is larger than specified, the buffer size determines the maximum amount of data to be transferred.

If the replication service does not have a version's payload data available, it checks if there is a placeholder for the replica. The placeholder contains the creator of the version, to whom the service sends a remote request for the version. The placeholder could have failed since the creation of the replica. If the connection request to the replica creator is unsuccessful, the replication service tries to contact the holders of backup replicas, which are also stored in the replica metadata. If there is neither a replica nor a replica placeholder, the replication service queries the key-based routing for the creator of the object. In the worst case, even the creator has failed, but the key-based routing protocol ensures that a request for an object eventually arrives at the corresponding manager.

4.2.2 Versioned replicas

The interface between consistency models and object coherence is the identifier management for multiversion objects. An object access subsystem implements coherence protocols based on version identifiers. Consistency models specify the compatibility of object versions in terms of version identifiers. Management of version identifiers is similar to handling of commit identifiers as presented in Subsection 3.1.2. The difference between these two kinds of identifiers is that commit identifiers pertain to a global consistent state, whereas version identifiers are tied to single objects.

Version identifier handling must be efficient in storage and time requirements and consists of two fundamental operations. The first operation is concerned with the generation of identifiers. When

creating a new version, the storage requests the version management to generate a new identifier for the version. The second operation interfaces with consistency management. If presented two or more different version numbers, the identifier management must be able to decide which is the more recent version.

The selection and comparison of version identifiers affects the efficiency of the two base operations. Arbitrarily chosen version numbers are most flexible, but the generation of version identifiers needs to know which version numbers are still free, and the decision about the most recent version involves much communication. Alternatively, the identifier of the node that produces a version can be encoded into the version identifier. Incorporating node identifiers in version numbers introduces single points of failure, and the decision about currency involves communication with the encoded nodes, for example using expensive two-phase commit. The third alternative are monotonically increasing version counters per object. In order to generate a new version identifier, the most current version of an object must be known. The storage can determine the most current object in an efficient and scalable manner by using an appropriate convention. One convention would be that each older version be marked as invalidated. An alternative convention would be to ensure that the manager node for the object can be contacted fast, for example using a key-based routing technique, as discussed in Chapter 2. The big advantage of increasing version identifiers is that comparing two identifiers is a fast local operation.

The preceding reasoning implies a specific design for version management. Monotonically increasing version counters enable efficient operations for version creation and comparison, but they are neutral to consistency management. The compatibility of different objects's versions is solely defined by consistency models. Wraparound of version identifiers can be handled with the protocol described in Subsection 3.1.2.

A practical implementation of object version management benefits from additional conventions. Special identifiers can denote an unknown or undefined version number or a zero-filled object version. Each replica stores in its metadata the version number when it was created and the version number when it was invalidated. The latter is initialized with the special undefined value if the version is still valid.

The conventions make interfacing with the object access and storage consistency modules simple and effective. Read and write operations on replicas return the actual version read respectively written. Specifying the undefined value for a version when reading an object retrieves the most recent version that is known locally. This convention is useful if the consistency module does not know the version state of an object, because it has not encountered the object before. Specifying the undefined value when writing an object updates the last version. The latter feature is needed in order to support local commits (see Subsection 3.1.3).

4.2.3 Deleting obsolete replicas

The discarding of obsolete replicas requires special attention, because object availability and spare memory use are potentially conflicting goals. On the one hand, the storage must guarantee the availability of an object version at least as long as some node can reference the object. If a node accesses a version whose replicas have all been lost, the storage must either create an exception to be handled by the application or rollback the state of all nodes to a consistent snapshot. Even after deleting an object, its data and metadata must remain available as long as references to the object can possibly exist. On the other hand, storage capacity is finite, such that versions that are not needed anymore should be discarded before the replicas exhaust the storage capacity. Conservative replica management that keeps versions for long time spans can eventually run short of memory, such that creating a new version becomes expensive.

A storage service that is decoupled from consistency management cannot determine on its own whether a replica is still needed or not. The decision about which replicas are obsolete depends on the storage consistency that is in effect for the object. The service must therefore fetch the information it needs from the consistency model.

The outlined approach to replica deletion can be used with any consistency model. To keep track of which replica can be deleted, the consistency model must report which replicas can still be accessed.

name	signature	semantics
`wait→version_id`	`object,offset, value,comparator`	wait for a replica of the object fulfilling the predicate

Table 4.2: Replication wait API

This information is directly available from the transactional consistency model described in Chapter 3. The procedure to determine discardable versions in transactional consistency has been described in Section 3.1.2. In contrast, weaker consistency models must provide callbacks for the replication service to determine whether a specific object is still needed or not. The strength of the used consistency model has a twofold impact on replica deletion. On the one hand, if the consistency model does not keep track of used replicas, finding out which replica might still be accessed can incur a high overhead. On the other hand, weak consistency models such as eventual consistency often tolerate accessing newer replicas.

Although it is generally hard to determine the versions that nodes will potentially access in the future, stronger consistency models and structured communication reduce the problem's complexity. Monotonic-read consistency guarantees that a node will never access an earlier version of an object, such that it suffices to know the latest version accessed by each node [182]. A central component that registers object versions simplifies the communication pattern. It maintains the necessary information which versions exist and which versions the nodes see and communicates with them regularly. A central component can impede the scalability of a distributed system, but history pruning can be done lazily. If object versions are stored in a distributed manner, lazy dropping of unnecessary versions avoids frequent communication for cleanup.

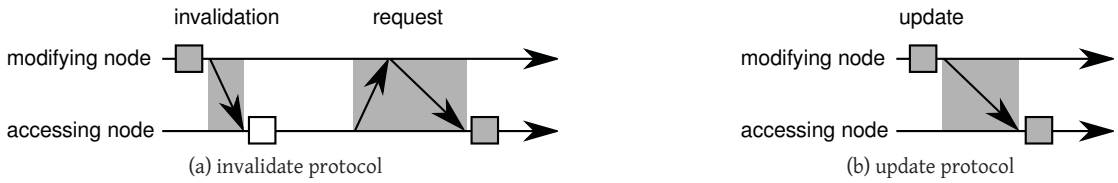
4.2.4 Synchronization on object content

Distributed applications that are neither event-driven nor operating continuously need to regulate control flow using synchronization primitives such as semaphores or monitors [101]. Busy waiting is an expensive and inelegant solution that does not integrate well with consistency management. The description of a replication service has focused on efficient handling of object content so far. This subsection details how to implement a synchronization primitive in a replication service.

The described synchronization mechanism, whose API is defined in Table 4.2, offers a `wait` operation similar to Hoare's condition variables [101]. A call to the `wait` function passes an OID, an offset within the object, a target value to wait for and a predicate that specifies any comparison operator for integer numbers. The call blocks until the object specified by the OID has a version for which the object content at the specified offset compared to the target value fulfills the predicate. On each modification of the respective object, the storage service checks whether the condition is fulfilled. If it is, it terminates the waiting and resumes execution of the application. The check integrates well with transactional semantics where modifications are broadcast to each node. If the implementation of the consistency model does not guarantee to notify each node about each modification, a publish-subscribe service can provide the necessary notifications, as described in the subsequent subsection. Chapter 6 gives examples on the usage of the `wait` operation.

Except for the use of a predicate to pause execution, the `wait` function has different semantics than the equally-named function on traditional condition variables. The definition of traditional condition variables requires them to belong to a so-called monitor object, which serializes execution of all the monitor's methods. If access to the monitor is not serialized, there may be race conditions between waking up from the `wait` function and simultaneous modification of the condition variable.

The Pthreads library, which implements a wait/signal mechanism without monitor objects, circumvents the race condition by associating each condition variable with a binary semaphore (mutex) that must be acquired before waiting on a condition. The mutex will be released atomically just before block-



(a) invalidate protocol (b) update protocol

Figure 4.1: Replica coherence protocols

ing and re-acquired atomically just after unblocking, such that signaling the condition when holding the same mutex the race condition described above [32].

The Pthread solution against race conditions cannot be directly applied to a large-scale replication service because of two reasons. First, pessimistic synchronization using semaphores would counteract the high scalability of the replication service. Second, the replication service is orthogonal to data consistency, such that it cannot decide on its own whether an object's content is still valid or not. On this account, the replication service's `wait` function does not guarantee that the predicate still holds when returning from the call. If an application needs to ensure the validity of the predicate, it needs to check the object's content again. Many use cases do not need this extra check, because it suffices to know that the object has fulfilled the predicate already.

4.3 Streaming updates versus invalidates

Replication achieves availability at the cost of increased storage requirement. Each replica of an object requires the same amount of storage as the original object. However, in order to increase reliability and to reduce the latency of object accesses, a storage service usually creates more than one replica. Full replication requires space linear in the amount of objects and nodes, such that adding more nodes to a fully replicated storage does not increase the overall capacity of the storage. Therefore, implementations limit the degree of replication. For most access patterns, neither pure on-demand replication nor pure update replication are optimal with respect to storage requirements and runtime. Therefore, a storage service should implement an adaptive replication strategy, taking into account prospective object access patterns, fault tolerance, the storage capacity of individual nodes as well as the runtime and bandwidth required for replication.

This subsection describes the design of such an adaptive replication strategy that switches dynamically between on-demand and update replication. The first subsection describes a mechanism to use different coherence protocols side by side. The second subsection details how the adaptive storage service monitors and predicts object access pattern. The third subsection presents a combination of adaptive replication and a publish-subscribe system for object updates, and the forth subsection outlines further optimizations to reduce object access latency.

4.3.1 Replica coherence

A versatile storage service must be able to adapt replica coherence to changing object access patterns. Coherence protocols ensure that only consistent data becomes visible [98]. The following text assumes a replication service that is orthogonal to consistency handling, as described in Chapter 3. Therefore, the replication service can decide for each modified object which coherence protocol to apply. It can send its peer nodes either an invalidation notification or an update notification that contains the current object content (see Figure 4.1). In case a peer receives an invalidation message, it will request the updated object version before accessing the object. In case it receives an update message, it stores the up-to-date version in the local replica.

For most applications, object access patterns are rather complex, such that a static decision about the number and location of replicas to create does not achieve optimal performance. A smart repli-

cation protocol can exploit the tradeoffs between invalidate and update protocol in order to adapt to changing access patterns during runtime of an application. The prediction of future object accesses is more precise if the past access patterns are known. Therefore, the storage service must keep track of which node accesses which objects.

The decision whether an update should be sent to a certain node or not depends on the cost of updating the replica and the saving of a replica request. Updating the replica requires the creator of a new version to send a message. If the replication only aims at reducing access latency, it suffices to update the replica with an asynchronous message, but for fault tolerance, the update must be send using a synchronous message. The replication also increases the network load and the storage requirement at the replica's recipient. Once the recipient accesses the replica, it does not need to request the object from the node who created the last version, an action which would block the requesting node until the replica arrives. Usually, the storage system cannot predict reliably based on past access patterns whether a replica will be accessed by a node before the next version is created. Therefore, it must try to predict future object accesses. The predictions can either be based on hints by the application, which requires applications to support the storage management, or based on registered object access patterns.

To realize the flexible replication strategy, the storage service uses a generic replication mechanism which ensures that replicas are created reliably. When creating a new version, the replication mechanism pushes a predefined number of copies synchronously to fixed nodes. Furthermore, it pushes additional copies to other nodes which will probably access the version. The decision to which nodes to replicate is supported by monitoring information. If a replica is not yet available at a node that accesses an object, the object it replicated on-demand to the accessing node.

Replication increases availability, but it also aggravates storage requirements. A second motivation for replication is to distribute load in the storage system. If a storage node creates or accesses lots of objects, it can eventually run out of local storage for replicas and their metadata. In these cases, the node should delete replicas that were created to reduce access latency only, but it must retain replicas that ensure the fault-tolerance of the system. The storage can swap local replicas out to peer nodes using the push-based mechanism. Alternatively, the peer nodes can retrieve replicas from the overloaded node using pull-based replication. The overloaded node can delete its local replica if it is sure that his peer has safely received the version, for example after receiving an acknowledgement message or by using a reliable communication protocol.

Wherever the storage service creates replicas, it must be able to locate them afterwards, for example in case of a failure or to replicate an object. The information where replicas are stored can be collected using different techniques. One option is for nodes requesting replicas to publish the replicas they have regularly. The alternative is to let nodes that create a replica inform their peers about which replica they have created where. The latter technique is well-suited for scalable systems having modest update rates, and it can be combined in a straightforward manner with a consistency protocol that sends update notifications, such as transactional consistency (see Chapter 3).

4.3.2 Access correlation and prediction

Object access prediction is based on monitoring and analyzing recent accesses. The techniques used must be both flexible and efficient. Only a flexible access prediction policy is superior to manual code annotations. An inefficiently operating prediction process could in the worst case cancel the performance gains through access prediction. In cases where it is difficult for a self-adaptive system to identify access patterns, manual code annotations help replicate objects efficiently at the cost of increased programming effort.

Object access monitoring

In a distributed storage system, access monitoring must be decentralized, because tracking all access at a central location does not scale well. The monitoring service must avoid imposing a messaging overhead even for highly dynamic access patterns. Therefore, an efficient implementation of the service must store access information locally where possible.

access monitor on node A	
OID	NID: counter
01	B: 5
10	C: 2
11	B: 4, C: 3

(a) state during local accesses

access monitor on node A	
OID	NID: counter
01	B: 6
10	C: 2
11	B: 5, C: 3

(b) state after node B requests objects 01 and 11

access monitor on node A	
OID	NID: counter
01	B: 5
10	C: 1
11	B: 4, C: 2

(c) state after aging

Figure 4.2: Accesses and aging in the access monitor

```
update_access_monitor(readset, writeset)
{
  foreach entry in readset
    monitor[entry.node, entry.oid]++
  foreach entry in writeset
    monitor[entry.node, entry.oid]++
}
```

Figure 4.3: Code for incrementing entries in the object access monitor

An instance of the distributed monitoring service executes on each node and records accesses to local objects by remote nodes. The access monitor can map OIDs to lists of associated NIDs using a local hashtable. The fast object lookup in $O(1)$ allows efficient computation of which objects shall be sent as updates to which nodes. The distributed access monitors are empty after startup of a node, and they are not synchronized between nodes during execution, thus they do not require global sharing of replication information.

Each entry in the hashtable represents a distributed object and stores a list of nodes that have accessed the object recently (see Figure 4.2). The entries of the hashtable determine which remote nodes will get an update after the local node modifies the corresponding object. If node A modifies object 10 in the example of Figure 4.2a, it will send the new version to node C, but only an invalidation of the previous version to node B. Hashtable and list entries should be created lazily in order to avoid excess storage consumption. Deletion of list entries can also be done lazily, which allows to batch cleanup operations.

On the first object access by a remote node, the remote node is added to the the access monitor's node list, if the remote node is not contained yet. For each remote access, the monitor increments the respective entry in the hashtable, as shown in Figure 4.3. Remote accesses are registered when receiving commit notifications or *sync* messages from remote nodes. Figure 4.2b shows the access monitor of node A after node B has requested versions of objects 01 and 11, which incremented the respective access counters.

Given that access patterns change over time, entries must be deleted from the data structures in order to avoid them growing monotonically. The deletion of entries can be accomplished using aging. All node entries are periodically recomputed by an aging routine running in a preconfigured time interval. The monitor should adapt the interval length dynamically depending on the number of objects stored in the hashtable. Figure 4.4 shows pseudocode for the aging of entries and the adaptation of the timer interval. If the table is highly populated, the aging routine is executed more often, resulting in fast decreasing access counters. If the monitor is filled only moderately, the recomputation executes in larger time intervals. In addition to aging, extensible hashing helps accommodate a growing number

```
age_monitor_entries()
{
  if (monitor.size > 0)
  {
    decrement = max(accesses.size / monitor.size, 1)
    foreach entry in monitor
      entry -= decrement
  }
  if ((monitor.size > monitor_size_upper_threshold) && (timer_interval > .5))
  {
    timer_interval -= .1
  }
  else if ((monitor.size < monitor_size_lower_threshold) && (timer_interval < 2.5))
  {
    timer_interval += .1
  }
}
```

Figure 4.4: Code for aging of entries in the object access monitor

of objects. Experiments with different realistic workloads showed that an interval range from 0.5s to 2.5s is a good choice for a throughput of several hundred transactions per second.

The aging routine decrements all access counters by the sum of all object accesses since the last aging run divided by the number of object elements stored in the access monitor. To ensure progress of aging counters, the routine enforces a minimum aging step of one. The monitor can delete a node if its counter reaches zero. An entry in the hashtable can be deleted if either the corresponding object has been deleted or if it does not contain any nodes. Figure 4.2c shows the access monitor of node A after a run of the aging routine. Assuming that there have been 2 object accesses since the last run of the routine, all three counters have been reduced by one.

Access prediction

The information collected by the distributed access monitor allows to predict future accesses in order to replicate updates actively. To this end, during a modification of an object, the storage service looks up the nodes that have recently accessed the object. These objects will probably access the object again soon, and they should favorably receive an update for their replica in advance. Of course, it can happen that the object is not accessed before it is modified again.

In accordance to Chapter 3, the smart replication protocol described here assumes a transactional storage service with a central node that validates transactions. Generally, the protocol can be applied to any consistency protocol, but the sending of commit notifications simplifies spreading updates. Each node sends its commit requests including read set and write set to the coordinator, which validates the transaction and replies with an OK or Abort message. Furthermore, the coordinator sends commit notifications, which includes only the write set, to all nodes. In addition to central validation, nodes perform an on-the-fly validation of any locally running transactions when they receive a commit notification, which reduces the amount of remote validations for transactions that are doomed to fail.

With a smart replication protocol, each node sends its commit requests to the validating node, just as with an invalidation-only protocol. However, the coordinator does not know which nodes get updates and which get invalidates. Therefore it leaves sending commit notifications to the requesting node. After having received an OK from the coordinator, a node sends out object updates using the access monitor. For each object in the write set, it checks whether the object is registered within the access monitor. If it is, the node sends updates to all peers in the node list. Updates are sent using separate network packets, as the write set of transactions is unbound. However, sending several messages over a stream-based communication protocol such as TCP/IP has only little overhead. Nodes receiving updates store them in their local replica manager; local access for the time being prohibited. After the update phase is finished, the node sends out commit notifications to all other nodes. The nodes that receive

write sets process a local validation, and their replica managers grant access to the previously received updates.

The distributed sending of commit notifications relieves the coordinator from sending out many network packets for all transactions in the global system. Furthermore, this approach also allows a committing node to send updates before the commit notification, such that invalidated remote transaction always have an up-to-date copy in place after restart. In addition, this ordering ensures that each object that receives updates is always up-to-date when an update is being published, because the publication of a new version and the arrival of the corresponding update are one atomic operation.

The node that validates transactions can also participate as a storage node. In that case, sending intended updates along with validation requests can reduce access latency for some applications. The measurements in Chapter 8 show that this is especially effective if conflicts on the corresponding objects are infrequent and the validating node accesses objects with a high probability.

Manual code annotations

In addition to access monitoring, the storage service can also support programmers in giving explicit replication hints during object allocation. Although code annotations about object accesses are less elegant than self-adaptive access prediction, a number of use cases benefit from explicit replication hints. For example, the access monitoring must gather information about object requests for a longer period of time. It cannot predict accesses for newly created objects. Similarly, the adaptive automatism cannot anticipate application logic in some cases.

Explicit hints are useful if access patterns are known to the programmer in advance. The creator of an object can request to always get updates for this object. Alternatively, the validating node can be specified to get all updates for certain objects. Another option is to request that nodes will always receive updates for the object. Chapter 6 gives examples of such applications. Distributed programming models such as MapReduce [59] perceive large objects as being partitioned and handled by certain nodes. Identifying these application-defined access patterns can be complex for the access prediction subsystem in the replication service, but the application can easily provide the replication service with access hints.

Several semantics for code annotations concerning replication are possible. First, the storage can offer an API for prefetching of objects. This prefetching API resembles the `madvise` system call implemented in the Linux kernel [183]. Like `madvise` and similar prefetching techniques, an application can benefit from the replica prefetching API only if it requests the replicas early enough. Second, the group of replica holders for an object can be specified during allocation or using a dedicated API. Third, the receiver of a replica can be specified when modifying an object. The latter techniques require the application developer to suppose which nodes are present in the system. Application code containing such assumptions is inflexible and has limited portability.

The replication hints are stored in the access monitor and can be changed or switched off during runtime by the program. Furthermore, manually added entries in the access monitor are marked in order to prevent the aging routine from deleting them automatically. They will however be deleted when the object itself is deleted. The latter is propagated to other nodes in order to allow them to detect object removals and to remove such objects from their local access monitor.

4.3.3 Publish-subscribe for object updates

An important design parameter for a distributed storage is how nodes receive information about object updates. So far, the discussion has assumed that the consistency protocol automatically notifies each node about modifications of all objects. With the transactional memory protocol described in Chapter 3 the validating node notifies its peers about modified objects by means of commit notification messages. The broadcast of commit notifications is well-suited for small to medium numbers of nodes and random probabilities of object accesses. However, in large-scale networks, groups of nodes tend to access only a subset of objects. This locality of accesses is caused by nodes working on related topics or any activity that makes them access the same data structures.

name	signature	semantics
`disable_replication`	`objects`	disable sending the specified objects to remote nodes
`enable_replication`	`objects`	enable sending the specified objects to remote nodes
`is_replicated_remotely`	`object →boolean`	check the replication state of the specified object
`create_version`	`object,version=default, buffer`	update the last committed version of the specified object

Table 4.3: Replication API related to local commits

Consistency protocols unlike the previously mentioned transactional protocol implement partial replication of update information. If the consistency protocol does not broadcast modifications, the `wait` operation defined in Subsection 4.2.4 needs further support from the storage service. Without this support, a node could wait endlessly on an object for which it does not receive any updates. Local commits (presented in Subsection 3.1.3) are a typical example for this phenomenon: An object that is modified by local commits does not cause any commit notifications to be sent. Thus, the storage must be able to configure update notification automatically, without support from the programmer.

Notifications about updates can be implemented using a publish-subscribe system [28] using the following protocol. A node waiting for an object update needs to subscribe to the object's topic in the publish-subscribe system. Unless a modification is broadcast to all nodes, the storage generates a publish event, which is sent to all subscribed nodes. Publish-subscribe can be implemented efficiently on a key-based routing network [169]. In terms of the OID space management scheme presented in Section 2.1.4, an object's manager node keeps track of the subscriptions of that object.

4.4 Optimizations

The adaptive replication strategy described in the previous section can be combined with other techniques to improve the reliability and efficiency of replication. First, a special interface between replication and consistency module enables optimization for non-replicated objects. Second, an extended storage service can use the replication mechanisms to manage replicas for fault tolerance. Third, delta encoding provides a way to reduce the bandwidth requirement for replication.

4.4.1 Support for local commits

A replication service that is implemented orthogonally to consistency models can support local commits. Subsection 3.1.3 has defined local commits to change non-replicated objects. However, the support for local commits requires softening the strict separation between replication and consistency module. The extended interface for the replication module to support local commits is listed in Table 4.3.

In order to check if a local commit is possible, the replication module must tell the validation procedure whether specific object versions have been replicated or not. The check ignores invisible replicas such as backup replicas (see Subsection 4.4.2). Furthermore, the consistency module must ensure that the object versions are not being sent to other nodes while it is checking the replication state of all modified objects. Disabling the sending of replicas requires inter-module locking, which is generally not desired but in this case unavoidable.

In case a local commit is admissible, the commit procedure updates the data of the modified objects. Given that the modified objects keep their version number, the commit procedure could specify the last committed version number. However, the modified data differs from the last committed version, such that ambiguous content with the same version number would exist. It is more safe to update an existing replica by specifying a default version number.

4.4.2 Masking node failures using backup replicas

The notion of availability embraces access performance as well as fault tolerance. Chapter 1 has detailed that these goals diverge. Large-scale storage systems that are built to be fault-tolerant create several replicas [88]. However, the more replicas exist, the more nodes must the storage service contact in case of an update.

The replication service described above supports local commits for non-replicated objects. To benefit from local commits, the adaptive replication strategy tries to replicate objects only if these replicas will probably be accessed on another node. If a node crashes, the content of non-replicated objects is lost.

It is nonetheless possible to combine local commits and replica-based fault tolerance. To this end, a fault-tolerant storage service uses a category of backup replicas. These backup replicas must not be accessed during normal operation. When modifying an object during a local commit, a node sends backup replicas to its peers much like the usual replication procedure. If a failure is detected, the storage selects one of the remaining nodes as its replacement. Chapter 2 has proposed a replacement strategy to make the key-based routing mechanism route subsequent object accesses to the replacement node. After its selection, the replacement node collects the backup replicas and converts them to regular replicas.

4.4.3 Delta encoding

Section 4.3 has primarily focussed on the latency cost of replication. If many updates are being replicated in a network, bandwidth is another important performance factor. A bandwidth limitation constitutes a potential bottleneck for data throughput, because message congestion increases communication latency. Delta encoding, colloquially called diffing, is a technique to reduce the size of update messages.

Definition 12 *Instead of sending the full updated content of an object, delta encoding computes the difference between the just created version and the previous version of the object and transmits only data and position of the differing parts.*

Diffing of memory pages has been documented for the Munin DSM system [113]. Delta encoding is especially efficient if large parts of an object remain unchanged, such that a small delta message represents a change to a much larger object. Rogers et al. present the BTMD algorithm to compute minimal diffs efficiently [168]. The page twinning used by BTMD works well with transactional storage, which creates shadow replicas for modified objects (see Chapter 3).

An object store can benefit from delta encoding for update replication. The sender usually has retrieved the previous version before writing to it, so that the computation of deltas is a straightforward comparison. The receiver can restore the full replica from the delta only if it has the old version of the object available, and if the delta belongs to the direct successor of the old version. In case the first condition is not fulfilled, the receiver can only discard the update. The second condition can be broken by update messages that overtake each other. A limited number of overtaking messages is easily

handled using a buffer in which the replication service stores out-of-order delta messages until it can apply them in correct order.

4.5 Related work

The orthogonal design of replication and consistency has been investigated by the JuxMem project [17]. JuxMem is a large-scale data sharing service that builds on a peer-to-peer communication service. Its authors argue that the decoupling of fault-tolerant data management from consistency models allows for a clean design with simple implementation and various options for experimentation. The replicating storage service discussed in this chapter shares many of the goals and characteristics with JuxMem, including a P2P communication structure and transparent access to update-anywhere replicas. Beyond these characteristics, this chapter details MVCC including garbage collection of ancient versions, support for transactional consistency and smart adaptation of the replication strategy.

The most related work to this chapter is the research by Dash and Demsky on caching and prefetching techniques for DTM [57, 58]. In the same manner as the adaptive replication described in this chapter, their multiversion DTM supports transparent restart and rollback of local variables, and it distinguishes authorative, cached, and transaction-local replicas. Prefetching is controlled by the application using static hint expressions, similar to the code annotations we discussed in Section 4.3.2. In contrast to the proposed adaptive replication, replication activity is configured statically in the application code. The only dynamic optimization is to disable useless prefetching sites. Dash and Demsky also present two techniques to evict old replicas. A committing transaction invalidates objects using unreliable asynchronous messages. In addition, a cache that is nearly filled purges old objects. These simple mechanisms run the risk of discarding an object either too early or too late.

The ClusterSTM system is similar to the proposed adaptive replication mechanism in the interface and implementation of a DTM [31]. It also separates replication from transaction management, as discussed in Section 4.2. However, ClusterSTM does not replicate objects at all, but leaves caching to the application programmer. The non-caching approach allows to combine ClusterSTM with diverse application-level caching strategies but cannot benefit from synergies such as fast local reads and local validations, see Subsection 4.4.1.

Other distributed storages supporting transactions take static approaches to replication. Multiversioning for transactional memory has been described in the publication on JVSTM [38] that has been discussed in the previous chapter. Sinfonia leaves the placement and distribution of data to the application programmer [10]. The transactional cache TxCache stores results of computations on behalf of the programmer [151]. The Ballistic DTM protocol proposed by Herlihy and Sun does not replicate objects but moves objects to the nodes that wants to write to them. The Hyder transactional storage even replicates non-committed, possibly invalid changes to all other nodes, a strategy that makes its performance very sensitive to the conflict rate of optimistic transactions [25].

Several replication strategies for transactional memory have been implemented in the context of the Aristos project. These works focus on another aspect of replication in DTM: the consensus of nodes on intended transactions. The Polycert protocol by Couceiro et al. [55] proposes several modes of replicating update information. The protocol selects adaptively among several atomic broadcast based certification protocols. The certification strategy is orthogonal to the smart data replication we propose. Similarly, Carvalho et al. propose the STR protocol, which claims to achieve optimal manager-based replication for speculative transactions [42, 43].

TinySTM is a non-distributed STM that does not replicate objects, such that both access strategies, write-through and write-back, work on a single copy [78]. However, TinySTM optimizes the performance dynamically by adjusting the hash function used to associate objects and locks, the number of locks and the hierarchical locking configuration.

Most replicating large-scale storage systems such as GFS [88], Bigtable [44] and Percolator [148] create a limited number of replicas at fixed locations to improve fault tolerance. The RAMClouds project argues that replication to increase performance is too complex and does not pay off for servers connected over fast local network, so it replicates only to increase durability and availability of data [146].

Our evaluation of the adaptive replication technique does not back up this assumption.

The Cassandra storage system [122] partitions the object space in a stable manner despite node churn using consistent hashing with an order-preserving hash function. To achieve availability and durability, Cassandra implements several replication strategies. The basic strategy replicates each object to its coordinator's successors on the logical node ring. More advanced strategies add preference lists for rack-awareness and datacenter-awareness. The replicating storage described in this chapter makes replication decisions based on access statistics, not on logical or physical closeness, because it focuses on performance rather than reliability. Extending the replication strategy to take preference lists into account is straightforward.

4.6 Summary

The main contribution of this chapter is a smart replication protocol that adaptively switches between update-based and invalidation-based replication. The decision about the replication mode is dynamic over time and based on monitoring of object accesses. A second contribution has been the design of a replication system that is functionally separated from consistency models. With respect to issues resulting from the split design, this chapter has discussed the implementation of multiversion replica management, deletion of obsolete replicas and synchronization on object content as well as a number of optimizations to improve the performance of replication. The contribution presented in this chapter has been published in the proceedings of an international conference [158].

5

Adaptive conflict granularity

The locality of reference principle states that programs often access adjacent objects or reuse objects that they have accessed shortly before [63, 64]. Developers of storage systems take advantage of the locality principle to achieve better performance. Optimization techniques such as prefetching and caching reduce the latency of accessing adjacent objects within a small time interval (see Chapter 4). Other techniques allow to increase locality of reference. By allocating related objects adjacently (see Chapter 2) and by caching data in units larger than the typical object size, a storage system can enforce stronger locality of reference.

The performance of a distributed and replicated storage system depends to a large part on the frequency of conflicts between several nodes accessing the same object. Multiple objects contained in one unit can cause false positives when checking for conflicts. In order to benefit from locality where possible while at the same time avoiding false conflicts, a distributed storage system needs to implement a dynamic approach to adapt the size of conflict units. This chapter presents such a dynamic approach to control false conflicts in the context of DTM.

This chapter is structured as follows. The first section defines the terminology of conflict units and false conflicts. The second section describes the design and implementation of static false conflict avoidance for a distributed storage system. The third section extends the static avoidance mechanism to an on-line mechanism that adaptively avoids false conflicts.

5.1 Terminology

Concurrent accesses to replicated storage by several nodes can result in access conflicts. Chapter 3 has presented conflict detection and resolution from the viewpoint of storage consistency. This section refines the notion of conflicts, taking into account the sizes of objects and conflict detection units.

5.1.1 True conflicts

The situation of several nodes accessing the same object is called *true sharing*. In case of read-only accesses, true sharing implies temporal locality, such that replication improves the performance of true sharing situations. In contrast, if there is at least one write accesses, true sharing causes *true conflicts* with concurrent reads on the same object, as shown in Figure 5.1a. A node modifying an object must notify all other replicas to ensure their coherence. Coherence protocols such as update-on-write or

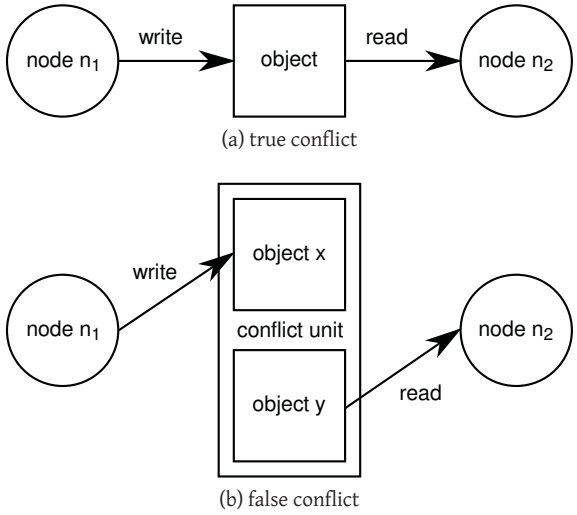


Figure 5.1: Types of conflicts

invalidate-on-write require network communication. Under strong consistency, where only one copy of each object exists, continuing true sharing makes the single copy thrash between the nodes. Using optimistic synchronization as implemented by memory transactions (see Section 3.1), transactions accessing the object while it is being updated must restart to access the updated replica. In general, the occurrence of true conflicts limits the possible degree of parallelism, resulting in a degraded performance.

The conflicts induced by true sharing situations can only be avoided by modifying the application. If the application is not yet well-parallelized, the developer can replace the single object by per-thread objects. Weakening the consistency requirements, e.g. by using snapshot isolation (see Subsection 3.2.2) can also mitigate true conflicts.

5.1.2 Conflict units

Most storage systems handle storage in chunks of equal size. These units are often larger than what an application typically requests at once. Transferring data in large units takes advantage of spatial locality. With a high probability, applications access colocated objects. To transfer a certain amount of data, using large transfer units needs fewer transmissions than using small units. Each data transfer usually involves a fixed storage overhead, such that large transfer units reduce the required bandwidth. They also reduce access latency, because the transfer of few large units is often faster than the transfer of many small units.

Handling data in fixed-size chunks is often more convenient than handling variable-size objects. The implementation of equally sized chunks is simple and efficient. Calculating the offset of a chunk in a buffer only requires multiplication of the chunk size with the current chunk index. Chunks once allocated can be reused without needing to reallocate because of insufficient size. Furthermore, the chunk size provided by the underlying data transfer layer allows for most efficient operation. For example, disk-based filesystems transfer data between volatile memory and permanent storage in disk sectors, and network transport protocols can achieve maximum throughput if they transfer data in the size of the maximum transmission unit. The memory management unit (MMU) that virtualizes physical memory detects accesses at the granularity of memory pages to simplify address translation. Despite using fixed-size chunks, a system can implement variable-size objects on top of the chunks. Small objects are created by splitting chunks using suballocation (see Section 2.3.3), and large objects are formed by aggregating adjacent objects (see Subsection 2.3.4).

Definition 13 *A* conflict unit *is a fixed-size storage chunk that serves for conflict detection.*

A conflict unit can contain several smaller objects, or it can be part of a larger object. The storage system can use the conflict unit size as the base dimension for storage replication to avoid translating between different object sizes.

The use of conflict units brings additional benefits for conflict detection mechanisms. Assuming good locality of reference, a data store can optimize the registration of accesses to colocated objects. It only needs to register the first access to an object within a conflict unit, such that subsequent accesses to objects in the same unit can proceed faster. The fixed size of conflict units allows to optimize checking for conflicts. For example, with a conflict unit size of 4 KB, the access validation mechanism can skip the lower-most twelve OID bits.

5.1.3 False conflicts

The aggregation effect of conflict units can also have adverse impact. In case there is no spatial locality between colocated objects, transferring conflict units instead of single objects increases the required bandwidth. Even worse, the access validation mechanism is unaware of the objects in a conflict unit, potentially causing the false sharing phenomenon. *False sharing* is a situation in which two or more nodes access distinct colocated objects and at least one node modifies an object [184]. The false sharing phenomenon results from nodes being unable to distinguish object accesses within conflict units. When two or more nodes access indistinguishable but different objects and at least one node modifies an object, all objects appear to be modified. Consequently, false sharing causes the algorithm checking for access conflicts to emit false positives.

Definition 14 False conflicts *are access conflicts that are caused by two or more nodes operating on distinct objects in a conflict unit, where at least one node modifies an object.*

In Figure 5.1b, node n_1 accesses only the first object, and node n_2 accesses only the second object, such that there is a conflict on the conflict unit as a whole, but not on the objects themselves. The validation mechanism only detects conflicts in the granularity of the conflict unit size. The aggregation of objects to conflict units causes false conflicts to appear where fine-granular validation would not detect a conflict.

Definition 15 *The conflict granularity of a storage object is the size of its conflict unit.*

Thus, the occurrence of false conflicts depends on the conflict granularity. Section 5.2 details that a storage can avoid false conflicts either by increasing the object size or by reducing the size of conflict units.

False conflicts limit the degree of parallelism and degrade performance just like true conflicts do. The impact of false conflicts becomes worse if they occur frequently. A single false conflict insignificantly affects overall performance. The advantages of spatial locality and less bookkeeping overhead can outweigh the slowdown caused by infrequent false conflicts. However, if a conflict detection unit is prone to false conflicts, it can become a limiting factor for performance and scalability of the application. As explained in Chapter 3, distributed optimistic synchronization is especially sensitive to conflicts. Given that fairness among transactions is difficult to ensure in a distributed system, false conflicts can increase the starvation risk of transactions. If the storage does not guarantee progress, a node can end up retrying a transaction that causes a false conflict over and over again.

5.1.4 Distinguishing false conflicts from true conflicts

An external observer, who is able to distinguish between objects and conflict units, can identify a false conflict without any problems. In contrast, the access validation mechanism sees a conflict for the whole unit and cannot tell a true conflict from a false conflict. Validation can only make assumptions about a conflict being true or false. These assumptions must be based on observations over periods of time with different conflict granularities.

Some access detection mechanisms are able to distinguish certain types of conflict. For example, page-based access detection on x86 processors has a default conflict size of 4 KB, but the effect of write operations can be analyzed a posteriori by determining the difference between modifications. Subsection 5.2.3 presents a method to combine fine-granular write-write conflict detection with snapshot isolation.

True conflicts and false conflicts are time-dependent phenomena. When access patterns change over time, true conflicts can turn into false conflicts and vice versa. A heuristic to distinguish false conflicts from true conflicts needs to revise its decision in time intervals. This insight motivates the following definition.

Definition 16 *The conflict rate is the number of access conflicts per time interval.*

The conflict rate depends on the number of accesses per node, on the number of nodes and on the conflict probability. All these parameters can vary over time, which makes conflict rate a time-dependent phenomenon. For a DTM that issues several hundreds of accesses per second, a typical conflict rate is in the order of magnitude of 10 per second or more. A high conflict rate suggests that either the program is not well parallelized or that the conflict unit is too coarse. If the access detection mechanism allows to scale the conflict unit size down, less false conflicts will occur after switching to a fine-grained conflict unit size. However, if the conflict rate remains high even for fine-grained conflict units, the conflicts are probably caused by true sharing. Section 5.3 presents a heuristic to dynamically adapt the conflict granularity to counteract false conflicts.

5.2 Static avoidance of false conflicts

As opposed to true conflicts, it is possible to avoid false conflicts. The basic idea in false conflict avoidance is to bring the conflict unit size more in line with the object size. The possible approaches are either to increase the object size by inserting padding or to decrease the conflict unit size. On the one hand, padding objects to a larger size causes space overhead. On the other hand, downsizing the conflict unit size can impact the locality benefit and thereby decrease performance. Some access detection mechanisms have restrictions on the minimum or maximum conflict unit size. For example, page-based access detection (see Chapter 2) cannot select a conflict unit size smaller than the minimum supported page size. This section focuses on the avoidance of false conflicts using page-based access detection.

Both application and storage system can counteract false conflicts. Assuming the application knows the conflict unit size, it can easily preclude false conflicts by increasing the size of objects it requests. Another option for the application is to reallocate objects that frequently cause false conflicts. The storage can pad object size as well. In addition, it can select a smaller conflict unit size by choosing fine-grained conflict units or by placing objects in separate conflict units. To retain the simple interface of the storage service, avoidance of false conflicts transparently with respect to the application is especially important. Therefore, the storage service must heuristically guess whether a conflict is a true conflict or a false conflict, and it must revise its decisions in a sensitive manner.

5.2.1 Multiview/Millipage address space layout

Although most modern processors support multiple page sizes (e.g. 4 KB and 2 MB respective 4 MB on x86 processors), operating systems usually do not allow applications to select the hardware page size. If objects are smaller than the page size, page-based access detection is prone to false conflicts, because accesses to different objects on the same virtual page cannot be distinguished. Delta encoding (see Subsection 4.4.3) permits locating write accesses at byte granularity. A writable page is always readable on x86 processors, such that diffing cannot preclude false conflicts, unless it reveals that a page has not been modified at all.

A simple approach to counteract false conflicts is to increase the size of objects to the hardware page size. However, placing each object in a distinct conflict unit trades exact access detection in for internal fragmentation. Although modern machines usually have plenty of physical memory available,

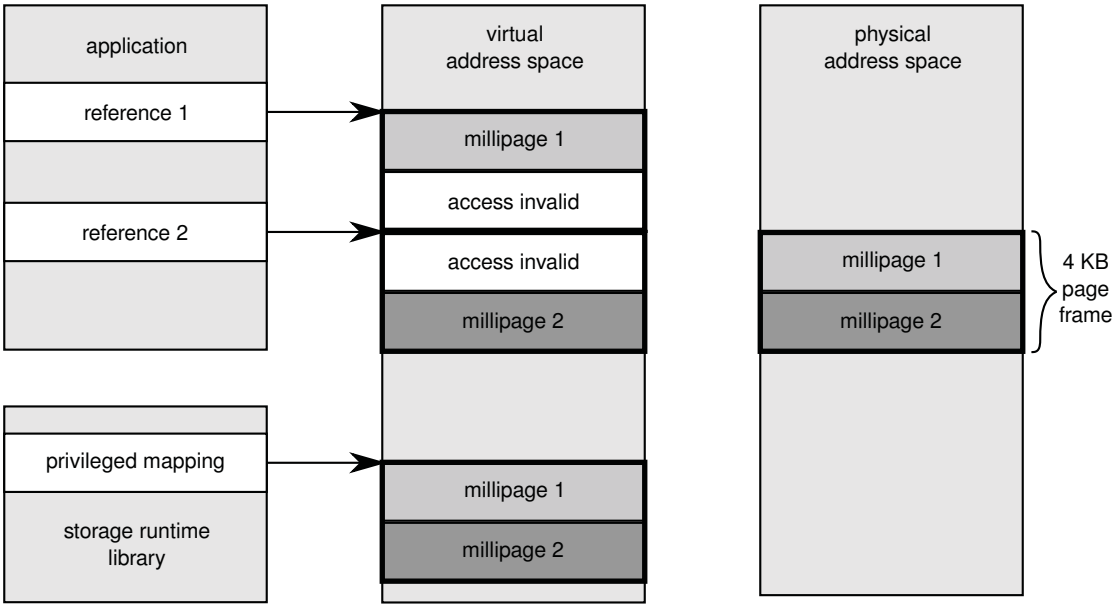


Figure 5.2: Millipage mappings ($n = 2$)

internal fragmentation can increase memory consumption by a factor of thousand in extreme cases, for example when wasting a 4 KB page for a 4 Byte object. Moreover, padding irrevocably eliminates the potential benefits of spatial locality, and it cannot adapt to different object usage patterns.

On systems with fixed conflict unit size, false conflict avoidance without memory overhead seems impossible at first sight. However, a storage can apply the Multiview/Millipage technique first proposed by Itzkovitz and Schuster [108]. Multiview simulates smaller conflict units called Millipages using a special virtual memory mappings. It maps a physical memory page repeatedly in the virtual address space and then hands out disjoint fragments of the same physical page to the application. Considering the huge virtual address space of 64-bit processors, the waste of virtual address space is negligible.

The Multiview technique effectively decouples page size and conflict unit size. If each conflict unit holds at most one object, every access uniquely identifies a single object. Therefore, Multiview is able to avoid false conflicts completely. In contrast to padding, Multiview causes little space overhead, because it uses physical pages efficiently. Each Millipage requires only a fraction of a physical page plus an additional entry in the page table. Given that Multiview enlarges the number of virtual pages accessed by an application, the technique increases TLB pollution. However, the reduced communication because of less false sharing compensates the prolonged address translation.

5.2.2 Implementation of Multiview

A Millipage region divides a physical page frame into 2^n disjoint Millipages. If the hardware page size is 2^p, one Millipage covers 2^{p-n} bytes. Each Millipage has a distinct mapping in the virtual address space, such that accesses to objects that reside on the same physical page frame are detected independently. A privileged mapping allows to circumvent access detection, for example to atomically update page content in multithreaded applications. If a privileged mapping is not available, the runtime system must stop all application threads in one address space before updating page content. Figure 5.2 illustrates the Multiview layout with two Millipages per physical page frame. Millipages and conflict units of different sizes can be used side by side.

It is convenient to implement Multiview by placing one region's Millipages on consecutive virtual memory pages, such that the region spans a range of 2^{np} bytes in which only a total of 2^p bytes belongs to valid Millipages. This simple convention about memory layout simplifies checking for invalid ac-

Purpose	POSIX system call	System V system call
Create a shared memory object	`shm_open`	`shmget`
Memory-map a shared memory object	`mmap`	`shmat`
Unmap a shared memory object	`munmap`	`shmdt`
Destroy a shared memory object	`shm_unlink`	`shmctl(IPC_RMID)`

Table 5.1: Shared memory operations specified by POSIX and System V

cesses. The application must reference objects in the first Millipage only through the first virtual page, objects in the second Millipage only through the second virtual page and so on. Equation 5.1 specifies how a valid address a must match a Millipage index i within a region that starts at address r.

$$a \in \left[\, r + i \cdot (2^p + 2^{p-n}) \,,\; r + (i+1) \cdot (2^p + 2^{p-n}) \,\right) \tag{5.1}$$

With the help of the Millipage layout convention, the storage can easily detect corrupt memory pointers.

To create identical Millipage mappings on all nodes, the Millipage granularity must be stored in the distributed metadata. In this way, Millipages of different granularities and full (non-Millipage) pages can be used side-by-side. When allocating an object, the storage automatically chooses the Millipage granularity coarse enough to hold the object. Each node can keep track of its unused Millipages, such that distributed communication during allocation or release of a Millipage is unnecessary.

The Multiview allocation scheme and the privileged mapping require multiple virtual mappings of the same physical memory page. The storage can construct memory mappings using POSIX or System V shared memory segments, which can be attached repeatedly to a single address space [183]. Table 5.1 contrasts the POSIX and System V operations to construct or destroy shared memory objects. Both sets of system calls have basically equivalent functionality. The POSIX operations have been introduced more recently than the System V operations.

Several synergies exist between memory transactions (see Section 3.1) and the Multiview approach. First, Multiview eliminates transaction aborts that are caused by false conflicts. Second, Multiview speeds up shadow copy operations. When creating a shadow copy for a Millipage, the storage needs to backup only a fraction of a full page, at most one physical page frame for an entire Millipage region. Similarly, Multiview restrains the range to compare for delta encoding (see Subsection 4.4.3). Third, the privileged mapping allows multithreaded transactions in a single address space.

5.2.3 Write-write conflict detection at fine granularity

As explained above, page-based access detection allows read access detection only at page granularity. However, delta encoding can detect changes at byte granularity by comparing the original content of the page with the modified version. To detect all changes to a page, the storage must run delta encoding just before reverting the page to not being writable (see Subsection 4.4.3).

The granularity refinement is able to detect false write-write conflicts, but it cannot help on read-write conflicts, because the exact positions of read operations remain unknown. Some consistency models, for example snapshot isolation, can benefit from refined write-write conflict detection by proceeding as shown in Figure 5.3. The validation phase in snapshot isolation starts by comparing the local read set with remote write sets at page granularity. Afterwards, delta encoding computes a refined local write set, which is validated byte-wise against remote write sets. This approach is not prone to aborts due to false write-write conflicts. To save bandwidth, the storage can encode remote write sets using Bloom filters [29], as suggested by Couceiro et al. [54].

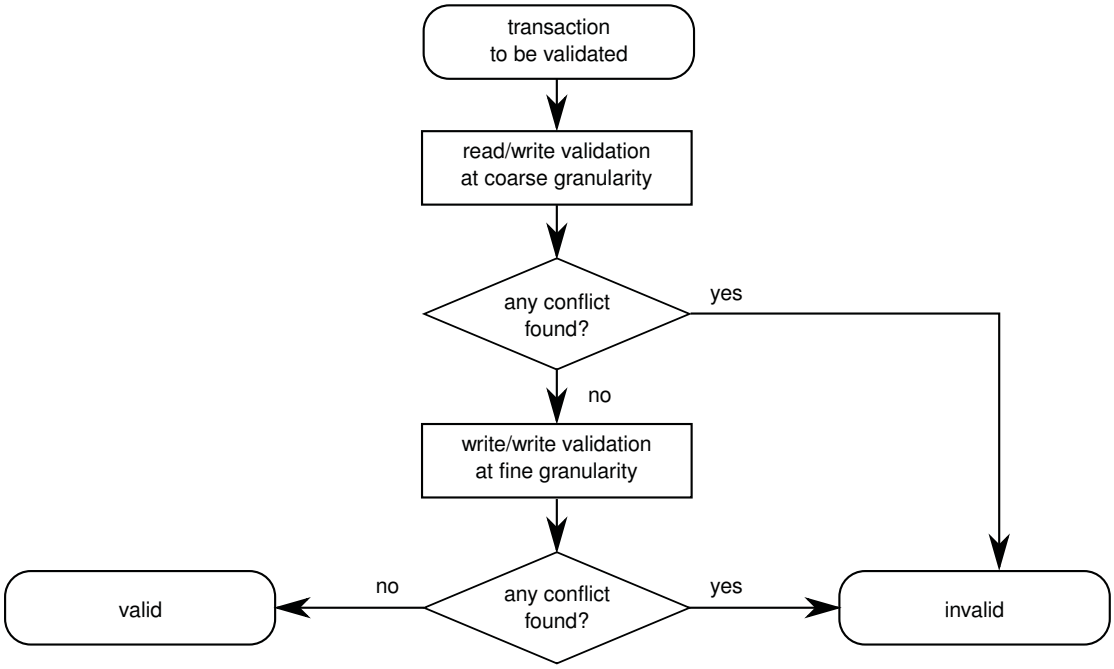


Figure 5.3: Validation with write-write conflict detection at refined granularity

5.3 On-line granularity adaptation using object access groups

The performance benefits of large conflict units concur with the increased risk of false conflict. Furthermore, a false conflict is a time-dependent phenomenon that depends on object access patterns, such that the decision for a conflict unit size needs to be revised from time to time. This section presents a heuristic to dynamically adjust conflict unit size for page-based distributed transactional memory (see Section 3.1). The heuristic combines a monitoring mechanism for object accesses with an adaptive policy for changing conflict granularity.

5.3.1 Monitoring of object accesses

A method to dynamically adapt the conflict granularity must coarsen and refine the granularity based on observed object accesses and transaction conflicts. On the one hand, if a set of objects experiences only few conflicts, aggregating them can improve the performance of data accesses and validation. On the other hand, an aggregated object that frequently causes conflicts can be split up to reduce the conflict rate and thereby improve the performance.

To avoid exponential state-keeping and limit memory overhead, the monitoring mechanism considers only objects located in the same Millipage region. These objects have been allocated by the same node during some time interval, such that a semantical relationship among these objects is likely. Furthermore, a single system call can set the access protection for a contiguous region of virtual memory, such that aggregating objects reduces the number of costly switches between user and kernel mode.

The dynamic adaptation mechanism bases its decisions only on local information in order to avoid network communication. Each node receives write sets from other committing nodes. Nodes need not transmit read sets, because remote read operations are not relevant to identify false conflicts using backwards validation (see Subsection 3.1.2).

Section 5.1.4 has already reasoned that a storage can only empirically distinguish true conflicts from false conflicts. The suggested adaptation mechanism monitors object accesses to determine whether Millipages should be handled separately or conjointly.

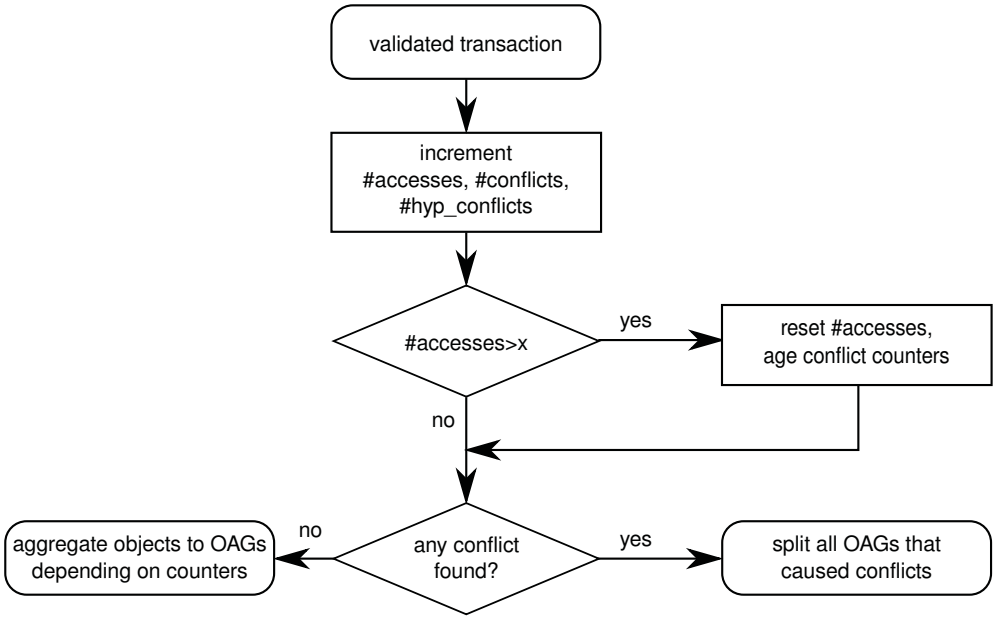


Figure 5.4: Dynamic adaptation of conflict granularity

Definition 17 *A Millipage region that serves as coarse conflict unit is called an* object access group (OAG)*.*

If a Millipage region is used as a coarse-grained object access group, the validation process cannot determine the conflicts for individual objects. Therefore, the adaptation mechanism can only refine the granularity using a trial-and-error method. Frequent conflicts for an OAG indicate that splitting the OAG might reduce the number of conflicts. However, if Millipages are handled as fine-grained individual objects, the monitoring subsystem can determine the existence of *hypothetical conflicts*.

Definition 18 *A* hypothetical conflict *is a set of accesses to different objects in a Millipage region that would cause a conflict if the Millipage region was handled as an OAG.*

A sharing situation with high spatial locality among objects in a Millipage region is characterized by few hypothetical conflicts. The real number of conflicts for a Millipage region is a lower bound for the number of hypothetical conflicts. If hypothetical conflicts are rare and a read set contains several objects from the same region, the adaptation mechanism combines the Millipages in the region into an OAG.

During the validation phase, transaction management determines whether a transaction conflicts with already committed transactions. In addition, for non-aggregated Millipage regions, the adaptive mechanism calculates whether OAGs would have caused hypothetical false conflicts. Each Millipage region counts the number of conflicts and hypothetical false conflicts. The counters are aged after a certain number of accesses which can be adjusted by the application. The experimentally determined default setting of halving the counters after 2^8 accesses achieved oscillation-free adaptation for the considered applications (see Chapter 8).

5.3.2 Adapting conflict granularity

The dynamic adaptation policy handles both the aggregation of objects to OAGs and the division of OAGs to objects with individual access detection. Figure 5.4 shows a flowchart for the algorithm, using counters for the number of accesses, the number of conflicts and the number of hypothetical conflicts (#hyp_conflicts). A sharing situation with spatial locality among objects in a Millipage region is characterized by few hypothetical conflicts. If hypothetical conflicts are rare and a read set contains several

objects from the same region, the adaptation mechanism combines the Millipages into an OAG. To avoid oscillation, OAGs are formed no sooner than several transactions after splitting the region. We determined empirically that a reasonable stabilization interval is equal to the number of Millipages in the region. An OAG that causes a conflict during validation is subject to false conflicts or even true conflicts. In either case, the handling as an OAG has worse performance than handling as individual objects. Thus, the adaptation mechanism splits the OAG immediately.

When aggregating objects into OAGs, it may happen that some objects in the group are not accessed during a transaction. Thus, the transaction's read or write set might contain false positives. For objects in the write set, generating a delta between the actual object and its shadow copy reveals whether the object has been modified (see Subsection 4.4.3). Given that a writable implies readable on most CPU architectures, transaction management must not ignore unmodified objects, but it can relocate them from the write set to the read set. For objects in the read set, it is impossible to detect whether they have actually been accessed in the transaction. False positives in read sets increase the probability of false transaction aborts but do not cause data inconsistencies.

The adaptive conflict unit size management is flexible and transparent for the application programmer. It relieves him of reasoning about data allocation and memory layouts causing false sharing situations. By providing an adaptive approach, large conflict units, fast validation and bulk network transfers are supported whenever possible. In case false conflicts show up, the adaptive management switches to a fine-grained Multiview consistency unit.

5.3.3 Hints for the application developer

Monitoring of object accesses can also assists the developer in identifying those objects that frequently cause conflicts. To prepare feedback to the developer, the storage aggregates conflict rates among all participating nodes and publishes conflict rates higher than a predefined threshold in the built-in nameservice (see Subsection 2.3.5). The published data contains information about the node and the function which have created the object and the functions that caused conflicts. The developer can extract true conflict hotspots from the name service either periodically or manually, for example before terminating the application.

5.4 Related work

False sharing effects have been observed and described several decades ago, mainly in the areas of multicore CPU caches and distributed shared memory. Torrellas et al. give a definition of the notion false sharing [184]. Bolosky et al. propose a cost component model for false sharing using heuristically selected interval lengths [33]. The tradeoffs involved in false sharing control have been described by Amza et al. [12].

The issue of false sharing has impacted the definition of several consistency models, for example scope consistency [105] and view-based consistency [103]. These models provide weaker consistency than strict or transactional consistency. The Region-trap library [35] combines pointer swizzling and virtual memory protection to trap accesses to individual objects, requiring region pointer annotation. Amza et al. [12] describe the dynamic aggregation of pages for lazy release consistency [113]. Our work has some similarities with ComposedView [145]. ComposedView provides transparent aggregation of small consistency units for sequential consistency, whereas our work integrates with transactional consistency. To tolerate false sharing, Thor implements split caching [176], which has been mentioned in Chapter 3. Split caching allows to merge modified objects on the same memory page, but it is incompatible with a hardware architecture where writable implies readable. The modern cloud storage service PNUTS has an adaptive consistency unit size, but it does not allow to switch the size dynamically [154].

The impact of false sharing on transactional memory has been discussed in the recent years, for example in the VELOX project [96]. Burcea et al. propose to vary access tracking granularity [37]. They define the *ideal granularity* to equal the conflict unit size that does not cause false conflicts. After analyzing different programs, they find that the ideal granularity varies greatly, and make suggestions how

variable granularity could be implemented. In contrast to our approach, the authors focus on per-object granularity that does not adapt dynamically to access patterns. Bocchino et al. [31] implement a DTM for large-scale clusters. They define eight design dimensions for their TM, one dimension is the static size of conflict detection units. To our knowledge, dynamic false conflict avoidance for transactional memory has not been analyzed and implemented previously.

5.5 Summary

The scalability of distributed storage under strong consistency requirements is limited by the rate of access conflicts. Avoidance of true conflicts requires to reengineer the application using well-known techniques, for example by replacing shared variables by thread-local variables. In contrast, the storage service can avoid false conflicts by forcing fine-grained access detection. Constraints of the underlying storage medium such as a minimum access granularity and writable-implies-readable can impede a fully transparent solution. In addition, a minimum access granularity can cause a high space overhead. This chapter has presented a solution for false conflict avoidance that does not require the application to intervene. Sharing is a time-dependent phenomenon, and the storage service cannot distinguish false conflicts from true conflicts precisely. Therefore, the presented false sharing avoidance uses a heuristic to switch the access granularity dynamically. The contribution of this chapter has been presented in a publication co-written by the author of this thesis [160].

6

A framework for extended MapReduce computations

Computing models describe common solutions for related problems. Application frameworks implement computing models in order to provide functionality shared by many applications. Particularly distributed applications benefit from using frameworks, because they are often faced with similar complex problems such as synchronization, fault handling and storage access.

This chapter presents a specific in-memory framework for distributed data analysis building on previously presented concepts. After motivating the use of in-memory storage for an extended MapReduce computing model, this chapter discusses job management using in-memory data structures. Finally it presents several applications that have been implemented on top of the framework.

6.1 In-memory storage for extended MapReduce

The previous chapters have discussed specific properties of in-memory storage. The use of in-memory objects enables distributed applications to use data-centric communication. It does not restrict the ways to build applications, because data-centric communication complements message-centric communication. Chapter 4 has already explained that regular data access patterns are an opportunity for the storage to achieve better performance. A programming framework can complement in-memory storage by enforcing certain data access patterns. MapReduce is a popular example for a computing model that can be implemented on top of in-memory storage as described in this subsection. Modern RAM sizes of 1 TB and more allow to place most computations in RAM, such that the capacity of in-memory storage is only an issue for extremely large data sets.

6.1.1 The MapReduce programming model

The MapReduce model, which has been suggested by the Google employees Dean and Ghemawat in 2004, is a programming model for distributed computations [59]. The model's workflow restricts the execution flow and data access of applications in order to enable a high degree of parallelism. The concept of Google's MapReduce has been inspired by the map and reduce functions in functional programming [123]. In contrast to functional programming, MapReduce is typically implemented by a programming framework that aims at simplifying imperative programming.

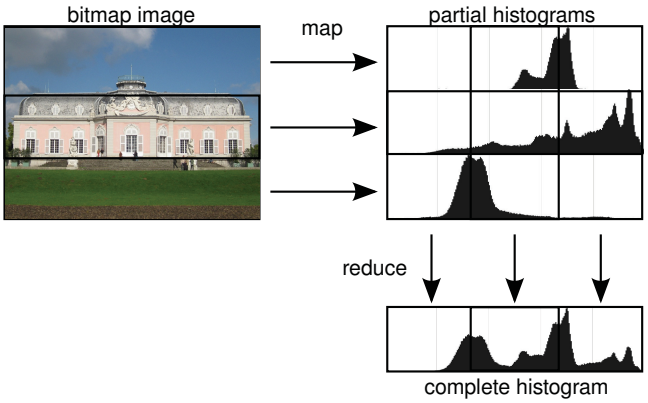


Figure 6.1: Data flow in histogram computation

An application that adheres to the MapReduce model consists of two phases: The map phase splits input data such that several worker nodes can compute intermediate results in parallel. The reduce phase transforms the intermediate results into the final result, again in parallel. A dedicated master node splits, shuffles and merges data and assigns jobs to worker nodes. Thus, the MapReduce model applies to algorithms of a certain execution pattern: two *embarrassingly parallel* phases executed in sequence.

An example for a typical application illustrates MapReduce's workings. Given a bitmap image, a histogram contains the frequencies of the individual colors (or intervals of colors) within the image. To parallelize the process of calculating a histogram, the image can be partitioned into disjoint sub-images. Each parallel activity computes a partial histogram of a distinct sub-image. Afterwards, the partial histograms are merged into a final histogram. Again, the merging can execute in parallel, if each activity walks through all partial histograms and extracts a certain partition of the color space. Figure 6.1 shows the data flow in parallel histogram computation.

The parallel computation of a histogram goes with the MapReduce computing model. The map jobs generate partial histograms, and the reduce jobs combine the partial histograms to the complete histogram. In the remainder of this chapter, we designate the original MapReduce model consisting of one map phase followed by one reduce phase as *single-pass MapReduce*.

6.1.2 Iterative and online MapReduce

Although there is a large number of parallel algorithms that can be modelled well with MapReduce, the data dependencies inherent in many other algorithms require a more flexible execution flow. Several MapReduce extensions for iterative and on-line execution partly repeal the restrictions and extend the MapReduce model to support data updates.

Some algorithms can be described as iterative execution of MapReduce, where either single phases or both phases in sequence execute iteratively. Iterative MapReduce makes approximate results available early and refines them with each iteration. Algorithms matching the iterative MapReduce model usually read constant input data and update intermediate or final results [74]. For example, the popular PageRank algorithm iteratively refines the relevance of webpages, an activity that can be modeled using iterative MapReduce.

Some other algorithms pertain to the on-line MapReduce model. This model runs with variable input data, which may either be an input data stream, or data that is frequently updated and requires recomputation of results [50]. Examples for on-line MapReduce are the on-line aggregation of page view statistics and the continuous processing of machine statistics. In contrast to iterative MapReduce, on-line MapReduce cannot be described as phases that execute sequentially. The following definition subsumes both iterative and on-line MapReduce.

Definition 19 *A computing model that executes sequences of map jobs and reduce jobs in alternating manner is called an* extended MapReduce model.

The relaxed definition of extended MapReduce allows repeated and asynchronous execution of map and reduce jobs as well as running map and reduce sequences iteratively. In contrast to original MapReduce, the iterative and asynchronous execution of extended MapReduce typically causes data dependencies within phases or reversely from output data of reduce jobs to input of map jobs. Unlike unstructured execution, extended MapReduce resembles MapReduce in that most data dependencies exists between map jobs and reduce jobs.

6.1.3 Data consistency and scalability of extended MapReduce

Single-pass MapReduce poses specific requirements on the underlying data store. On the one hand, the storage must be highly scalable in order not to constitute the bottleneck of the MapReduce system [144]. On the other hand, by virtue of restricted data dependencies in the MapReduce model, the storage needs not support updates and consistency [59]. To respond to these requirements, Dean and Ghemawat based their MapReduce framework on Google File System (GFS), which is optimized for write-once consistency and highly concurrent accesses [59]. GFS complements the access pattern of single-pass MapReduce with a client-centric weak consistency: It only guarantees that special append-at-least-once operations result in defined file content, whereas regular write operations leave the file consistent but potentially undefined [88].

Existing frameworks for extended MapReduce models trade simplicity in for flexibility. In order to support data updates, they propagate data updates using message-passing facilities that are not part of the storage system, for example publish-subscribe [74, 196] or streaming [50]. However, explicit messaging lacks the transparency of storage replication. It requires a higher engineering effort for the framework, for example by requiring an application to distinguish between static and dynamic data [74] or to specify to which node to pipeline intermediate data [50]. Furthermore, the additions limit data throughput, such that it can be used to transfer small amounts of data only [50].

For iterative and on-line MapReduce, which may update data, the storage service must ensure consistency. If the data store only supports modifications of single objects at once, race conditions can arise from modifications becoming visible in undefined order. Although a clever implementation can avoid most race conditions, such implementation is often complicated and error-prone. On the one hand, the storage service should support changing multiple objects in an atomic operation, such as a transaction on distributed storage. On the other hand, serializing all modifications of the distributed data store would limit the concurrency. Thus, the data store needs to ensure the serializability of operations where needed, while offering weaker access consistency where the application can tolerate them.

The implementation of distributed MapReduce can be based on an in-memory storage service like described in Chapter 2 instead of conventional MapReduce storage such as GFS.

Definition 20 *An in-memory MapReduce framework implements the MapReduce model based on a distributed in-memory storage as described in Chapter 2.*

In a transparently replicating in-memory storage for MapReduce, neither framework nor application need to specify which data to make available at which node. However, transparent replication implies that the in-memory storage identifies and possibly predicts accesses. Replication allows to cache data at those computing nodes where it has been used before, thereby reducing access latency (see Chapter 4). For example, if an animation shows objects moving in front of a static scene, only parts of the bitmap change between different frames, such that a major part of the bitmap can be cached. Another benefit of replication is that it can improve the data store's resilience in case of failures. The specific requirements on storage consistency depend on the application's data access characteristics. Because of the high integration of in-memory store and applications, applications can benefit from having precise control over the consistency guaranteed by the data store.

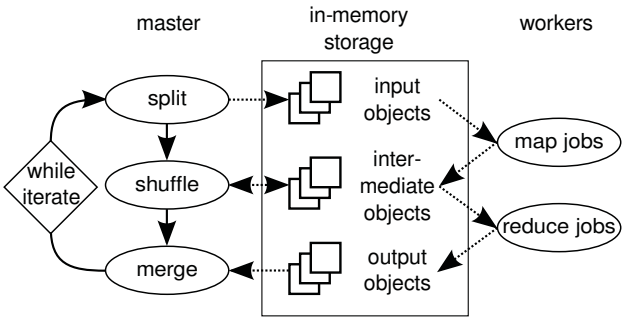


Figure 6.2: Execution flow in an extended MapReduce model

6.1.4 Extended MapReduce based on in-memory storage

Original MapReduce avoids data dependencies between workers in the same phase. Data dependencies only exist in the map phase between input date and intermediate results, respectively in the reduce phase between intermediate and final results, making both phases *embarrassingly parallel* [85]. Given that reduce jobs run only after all map jobs having finished, the beginning and end of each phase act as barriers, such that the execution flow implicitly synchronizes data. Thus, MapReduce simplifies synchronization at the expense of restraining data dependencies and control flow.

In contrast, extended MapReduce executes map and reduce phases in sequence, either one phase or both in alternating order, such that the results of one phase can serve as the input of the next phase. Figure 6.2 shows the execution flow in an extended MapReduce model. The extended model allows data dependencies between phases, but in many use cases most input data is not updated. On the one hand, these irregular data dependencies hinder processing iterative or on-line MapReduce problems with the single-pass MapReduce model. On the other hand, not all input data must be read again.

To avoid idle times, an iterative MapReduce framework may allow interleaved phases. In case of interleaved phases, reduce jobs may run even if some map jobs they do not depend on have not finished yet and vice versa. Interleaved phases contradict original MapReduce's implicit synchronization, such that the execution framework needs to track data dependencies. Stream processing using on-line MapReduce may cause jobs of any type to run as soon as data to be processed is available. Like interleaved MapReduce phases, stream processing requires explicit dependency tracking. To summarize, extended MapReduce models necessitate consistency handling that exceeds the barrier-like synchronization in original MapReduce. Therefore, a storage service that supports speculative synchronization such as a DTM (see Chapter 3) is appropriate for extended MapReduce.

Transactional consistency is useful for handling updates with low non-zero conflict probability. However, depending on the application, it can be too strong. If the application knows in advance which objects are prone to access conflicts and which are not, it can instruct the storage to synchronize and replicate objects more efficiently (see Chapters 3 and 4). The simplest approach is to run each map and each reduce job as a transaction. Transactional execution does not affect single-pass MapReduce except for the slight overhead of serializing non-conflicting transactions. For extended MapReduce, transactional storage synchronizes accesses. It executes map and reduce jobs speculatively and restarts them in case of access conflicts. Conflicts are caused by concurrent data updates and accesses. The structure of storage accesses in extended MapReduce results in low conflict probability. To improve performance, the application can reduce conflict probability further by running fine-grained transactions or by avoiding false conflicts (see Chapter 5).

6.2 Scalable and resilient job management

Like other MapReduce implementations, a framework for in-memory MapReduce must comprise work queue management for map and reduce jobs. The master node in an extended MapReduce framework

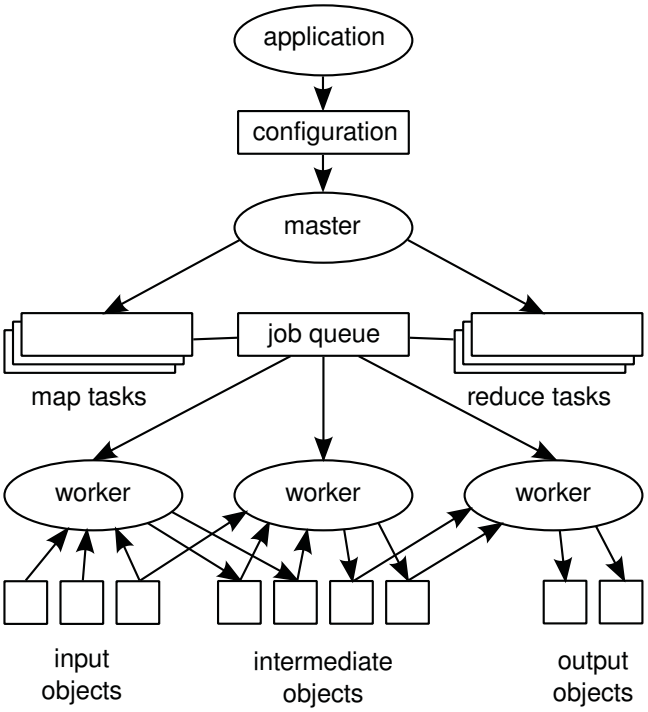


Figure 6.3: Objects accessed by in-memory MapReduce

schedules jobs to the worker nodes, such that resources are used efficiently. If the implementation is based on in-memory storage, the job management can store its data structures as memory objects. This section explains how the in-memory storage of job management structures simplifies assignment of jobs to nodes and job migration to implement work stealing.

6.2.1 In-memory job synchronization

Extended MapReduce requires a flexible job scheduling subsystem, which can build upon update notifications from the storage system, as explained in the following.

The fundamental concept in an in-memory job scheduler is the reification of all activities. The scheduler represents all processing nodes and all jobs as in-memory objects. Nodes have work queues attached, and there is a global work queue for jobs which can be scheduled on any processor. For most applications, the per-node queues are best suited, because they do not require global synchronization for extracting jobs from the queue. For a given number of worker nodes n, the first n jobs are assigned in round-robin manner, and successive jobs are distributed randomly to the nodes. The randomization defeats regular patterns in the execution times of jobs, which could lead to irregular distribution of workload. If the application uses the on-line MapReduce schema and job runtimes are long, the fraction of waiting jobs per worker is low. In this case, a global work queue is more appropriate. The accesses to job queue and data objects are depicted in Figure 6.3. For clarity of presentation, local job queues are not displayed.

The control flow of extended MapReduce contains a number of waiting situations, such as the master waiting for all jobs being finished, or the workers waiting for new map or reduce jobs. Busy waiting for a condition is inelegant and unnecessarily consumes network bandwidth, but an in-memory MapReduce implementation needs not resort to a message-passing approach. By storing the work queues in distributed storage, the runtime system can take advantage of a waiting mechanism to block until a condition becomes true. An exemplary wait function has been suggested in Chapter 4. The function `ecram_wait(object_id, value, comparator)` blocks the calling thread until the specified

object satisfies the comparison with the desired value. The comparator may be a test for equality, inequality, greater-than etc. Whenever the node receives an update for the object in question, it checks whether an updated value fulfills the condition. In that case, it resumes the previously blocked thread.

Besides storing descriptors of jobs waiting for being processed, each work queue also stores the number of jobs it contains. Using the `ecram_wait` function, nodes wait until at least one job is inserted into the work queue. Note that, by virtue of transaction properties, inserting a job into a work queue and incrementing the attached job counter is an atomic operation. Thus, the job scheduler is free of race conditions even without implementing lock management.

The described condition waiting mechanism allows the framework to synchronize execution of different MapReduce flavors. To execute conventional MapReduce jobs, the scheduler on the master node creates an integer object to hold the number of finished map jobs. It splits input data, enqueues the map jobs into the global queue and waits for the integer object to equal the number of jobs submitted. The worker nodes dequeue and execute the jobs, and afterwards each worker increments the integer object by one in a transaction. After having shuffled the intermediate results, the master node creates an integer object to numerate the reduce jobs, enqueues the reduce jobs and waits for the completion of all jobs.

Scheduling of iterative MapReduce proceeds similarly. However, the master node creates map and reduce jobs alternately until it has run a specified number of iterations, or a terminating condition is fulfilled. The worker nodes run exactly the same code as with conventional MapReduce.

Online MapReduce requires less job synchronization by the runtime system. The application can create map and reduce jobs as soon as input data is available, or it can create long-running jobs that check for pending data by themselves. However, long-running compute jobs bear internal state. But if internal state is completely stored in fault-tolerant memory, the application needs not take special precautions for cases of failures.

6.2.2 Load balancing

Job execution times in many MapReduce applications deviate significantly [13]. Differences in job execution time necessitate a way to distribute jobs evenly among workers. If job runtimes are known a-priori, the master node can compute a scheduling plan in advance. However, job runtimes in MapReduce are often unpredictable, such that the framework must implement a work stealing approach to balance load between per-node work queues.

Work stealing is a simple solution to avoid workers idling as far as possible if job runtimes are unknown in advance. The framework can implement work stealing as a distributed solution instead of requiring the master to monitor and re-assign jobs. If a worker finds that he is about to block on his empty queue, he scans the work queues of his peers for jobs to steal from them. Given that work queues are stored as shared objects, there is no danger of deadlocks or lost jobs. However, work stealing is a best-effort approach that cannot guarantee to deliver the shortest execution plan. In practice, round-robin job assignment combined with work stealing delivers sufficiently short schedules and avoids the complexity of more elaborate scheduling strategies.

6.2.3 Reliability and performance

MapReduce usually executes in a large-scale cloud computing environment, where resources are often paid based on their actual usage (pay-as-you-go principle) and failures are frequent. The large-scale computing environment of MapReduce necessitates handling unexpected node failures. In contrast to conventional MapReduce, extended MapReduce can contain stateful compute jobs, such that detecting missing output data does not suffice for fault tolerance [74]. In accordance to original MapReduce, we do not consider failures of the master node.

On the one hand, users may wish to add more computing nodes to speed up computation and access results earlier. On the other hand, Amdahl's law [11] limits the degree of parallelization, such that additional nodes may increase costs with negligible effect on processing speed. Therefore, the cloud platform needs to dynamically configure the participation of computing nodes.

To support extended MapReduce in a cloud-computing environment, the framework should support dynamic configuration depending on available resources and on the workload. The node management can be implemented using in-memory data structures much like the job management according to the following description. Each node is represented by a node information block, which serves to identify the node and contains the node's private job queue and the node's current status. The framework assumes a single master node. Worker nodes organize their node information blocks in a queue. Operations on the job queue and on the worker queue are implemented as transactions, thus the data structures are handled. When joining, a worker nodes enqueues its node info block and when leaving, it dequeues the block. To determine how to configure and schedule jobs, the job scheduler on the master node scans the worker queue periodically, for example before each map and reduce phase. When a job is scheduled to run on a node, the framework assigns it a timestamp. If an ancient timestamp identifies a lagging job, the node will be removed from the worker queue, and the job will be re-scheduled on another node.

The described node management supports nodes joining or leaving during runtime. Once map and reduce jobs have been scheduled for execution, the set of computing nodes is assumed to remain constant. Newly joined nodes are considered only at the next scheduler invocation. Unexpectedly leaving nodes are handled as failures. To automatically allocate resources depending on imposed workload, the framework must take into account the potential benefit and the overhead of reconfiguration.

6.3 Applications of the proposed framework

The framework makes implementing distributed extended MapReduce algorithms straightforward. The client nodes simply call `job_run`, passing an array of functions they are able to run as map or reduce jobs. The master node prepares storage descriptors for input and output, for example by mapping input files as shared objects. Then it calls `mapreduce` with the parameters described above. When the last worker sets the output descriptor's ready condition, the master node can write output objects back to the filesystem.

The map and reduce jobs can execute transactions to retrieve and modify distributed objects, such that they will always read the most current version, and stores are replicated to guarantee their durability. If data consistency is not an issue, because the structure of the control flow precludes race conditions, jobs can alternatively access objects without running transactions. Object accesses outside transactions allow the framework to omit serializability checks and thus increase the degree of parallelism.

6.3.1 Word frequency analysis

The classical example for MapReduce is word frequency analysis ("wordcount"), which determines the number of occurrences for each words contained in a text corpus [59]. The master node distributes the input documents evenly to the worker nodes, which in turn compute intermediate counts of the words in their subset of documents. In the shuffle phase, the master collects the intermediate histograms and generates an index table. Finally, in the reduce phase, the master directs the workers to sum up the counts for a specified range of words.

Iterative word counting makes intermediate results available early. Online computation of word frequencies allows for example tracking word frequencies for a collection of web sites. Instead of words, any other kind of items such as integer numbers and weblinks can be counted using the same algorithm.

Word counting can be implemented using the in-memory framework and in-memory tries (prefix trees) [117] (see Figure 6.4). The trie data structure almost entirely avoids false conflicts, because two insertions of words in a trie collide only if one word is a prefix of the other word or if both words are equal. During the map phase, each worker builds up his own tree, such that collisions are impossible and the map phase can run as a single transaction. In the reduce phase, each worker scans all trees for a certain subset of prefixes. The low conflict rate during the reduce phase allows to run long transactions.

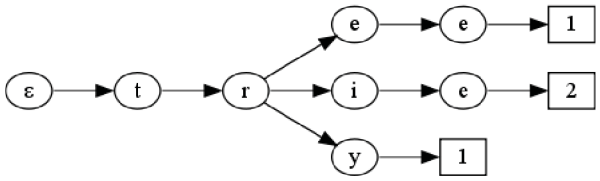


Figure 6.4: Trie representation of the three words “tree”, “trie” and “try”

6.3.2 Histogram

The computation of histograms for static images has been described above in Subsection 6.1.1. Histogram data of video streams can be computed by applying the extended MapReduce model. The image frames of the input video stream change with each iteration. Depending on how the histogram data is used by the application, it can store the output histograms in different storage objects, or it can store the output in a single dynamic histogram. In the latter case, the dynamic histogram will be updated by each MapReduce iteration.

Figure 6.5 displays the implementation of the map and reduce functions for histogram calculation. The map function iterates over all pixels in the image region specified as input. It stores the red, green and blue values of each pixel in the corresponding fields in the output array. The reduce function looks at each intermediate histogram in turn. In each intermediate histogram, it iterates over a range of color values and adds the values to the final histogram. All reduces write concurrently to the same output histogram, such that false conflicts between adjacent entries in the output histogram can occur. The implementation inserts padding between histogram entries to avoid false conflicts without complicating the source code. Placing each field of the output histogram in a separate object block costs less than 1 MB space overhead, but allows fully independent write accesses. The histogram map and reduce functions encapsulate all their storage accesses in a single transaction between the `ecram_bot` and `ecram_eot` markers. The code for configuring the framework to run the histogram application is printed in Figure 6.6.

6.3.3 Real-time raytracing

Raytracing transforms a 3D scene graph into a 2D image by tracing the path of light through the scene. The computation is embarrassing parallel, because an implementation can trace the rays through each pixel of the output image independently. In contrast to word counting and histogram computation, the parallelization partitions the output data, not the input data.

There are a several motivations for iterative raytracing. Raytracing can improve an image iteratively by replacing interpolated pixels with more exactly calculated pixels, or by increasing the accuracy of effects such as reflection and transparency. Real-time raytracing makes a case for online MapReduce, because nodes can update the image of a changing scene. When generating animations, the output bitmap size remains constant during iterative raytracing, such that the framework can reuse the storage for the bitmap. Therefore, raytracing makes a use case for adaptive replication.

The implementation stores not only the image bitmaps, but also the scene graph in distributed storage. The master node splits the 2D image plane into equally-sized partitions. Each node works on the pixels in the partition assigned to him during the map phase. The reduce phase collates all partitions to a complete image. The raytracer supports computing scenes at different resolutions and iterative rendering of dynamic scenes.

Either each worker node allocates his partial output bitmap himself, or the master node allocates the whole output bitmap before creating map jobs. The latter case works well with automatically adapting replication. In the former case, the adaptive replication is configured by the application to always send updates to the master.

The raytracing application benefits greatly from load balancing (see Subsection 6.2.2). The rendering of large, complex images often results in jobs that are much more compute-intensive than others.

```
typedef struct pixel
{
    unsigned char b;
    unsigned char g;
    unsigned char r;
} pixel_t;

void histogram_map(
  mapreduce_storage_t *in,
  histogram_t *out)
{
  ecram_bot(0, NULL);
  pixel_t *p;
  unsigned long pixels = 0;
  for (p = in->data + in->offset;
       (void *)p < in->data + in->off + in->len;
       p++)
  {
    out->r[p->r]++;
    out->g[p->g]++;
    out->b[p->b]++;
    pixels++;
  }
  ecram_eot(0);
}

void histogram_reduce(
  mapreduce_reduce_in_t *in,
  ecram_object_id_t out)
{
  // iterate over all intermediate histograms
  ecram_bot(0, NULL);
  int i;
  for (i = 0; i < in->nintermediates; i++)
  {
    ulong64_t first = 256 * in->id / in->nreduces;
    ulong64_t last = 256 * (in->id + 1) / in->nreduces;
    // iterate over a range within the histogram
    unsigned long long j;
    for (j = first; j < last; j++)
    {
      ulong64_t *data = out + ecram_get_block_size() * j;
      data[0] += ((histogram_t *)in->interm)[i].r[j];
      data[1] += ((histogram_t *)in->interm)[i].g[j];
      data[2] += ((histogram_t *)in->interm)[i].b[j];
    }
  }
  ecram_eot(0);
}
```

Figure 6.5: The map and reduce functions for histogram calculation

```
void histogram(
  ecram_object_id_t infile,
  ecram_object_id_t outfile)
{
  mapreduce_app_t *app =
    ecram_alloc(sizeof(mapreduce_app_t), NULL);
  strcpy(app->map_function, "histogram_map");
  strcpy(app->reduce_function, "histogram_reduce");
  app->input = read_input(infile);
  app->output_descriptor = outfile;
  app->split_size = sizeof(pixel_t) * pixels_per_map;
  app->intermediate_size =
    ecram_block_align(sizeof(histogram_t));
  app->final_size = ecram_get_block_size() * nvalues;
  app->nreduces = job_get_nworkers();
  mapreduce(app);
}
```

Figure 6.6: Code to configure in-memory MapReduce for histogram calculation

Therefore, the application can take advantage of load balancing by creating more jobs than workers. It could sort the job queue according to the estimated runtime of each job, such that long jobs are started first.

6.3.4 K-means clustering

The heuristic k-means clustering algorithm aims at partitioning points in an Euclidean space into k sets under the constraint that the distance between nodes and cluster centers is minimized [198]. The well-parallelizable algorithm has been implemented as an instance of MapReduce [195] and as an application for transactional memory [39].

The algorithm chooses the k initial cluster centers randomly. Each map job calculates the distance between one point and the k cluster centers. The reduce jobs then determine for each point which cluster center is nearest. If necessary, they reassign the point to the new center and adapt the positions of the cluster centers. Next, the master node sums up the position adjustments of the current run and decides whether another iteration is useful. The iteration of distance calculation and cluster center reassignment is repeated until the improvement of the adjustments is below a given delta. The number of iterations is also limited by a predefined number to avoid oscillation. Iterative k-means clustering allows accessing approximate results fast, with more precise results available later. Figure 6.7 exemplifies a k-means computation with $k = 2$ in four iterations. In a two-dimensional plane, data points are shown as squares, whereas cluster centers are shown as circles. The algorithm terminates, because the improvements in the third iteration are rather small. In the implementation, the reduce jobs track the movement of cluster centers. The post-iteration function checks whether the movement is less than the requested limit, and then stops the iteration by setting the `iterate` variable to false.

6.3.5 Lee's routing algorithm

Lee's algorithm solves the problem of finding shortest routes between sets of source and destination coordinates. For example, the algorithm serves to find routes in a maze or to layout digital circuits, where connections should be as short as possible to save material and achieve low signal propagation time. The algorithm takes a dynamic programming approach. A breadth-first search expands from the source field to more remote fields and tags each field with its distance to the source, until it finally tags the destination field. Subsequent backtracking from the destination to the source identifies the fields on the shortest path.

In a parallel implementation, all nodes determine shortest paths in a shared grid for given source and destination points. The nodes work on a shared queue, which is the major point of contention in

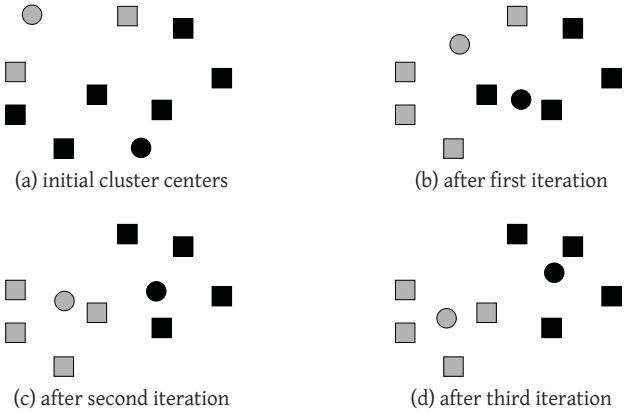


Figure 6.7: Four iterations of k-means computation

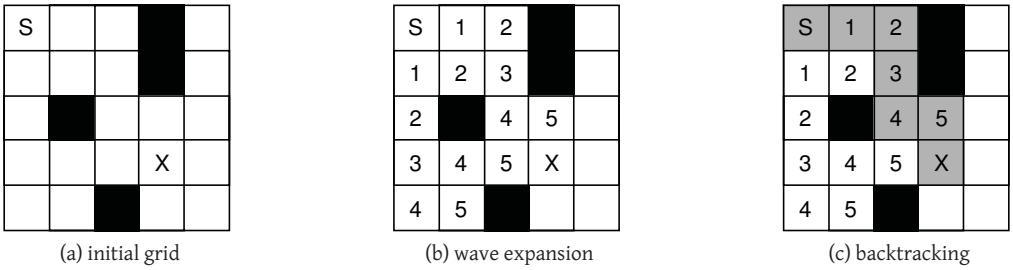


Figure 6.8: Lee's routing computation

the algorithm. The set of source-destination pairs is read from an input file into the queue. Evidently, a transactional implementation runs one or several routing attempts in a transaction.

Lee's algorithm has a large read set during the wave expansion phase. The backtracking phase, which works on a shared grid, causes conflicts with transactions that are routing paths crossing the one to be written into the shared grid. In that case, conflicting transactions will abort and restart from the beginning. Transactional implementations of Lee's routing algorithm have a considerable conflict probability [15]. Figure 6.8 presents the wave expansion and backtracking steps in the Lee algorithm.

6.4 Related work

The proposed in-memory framework implements an extended MapReduce model similar to Twister and Hadoop Online Prototype (HOP). In contrast to the framework presented here, Twister promotes data updates using multicast and publish/subscribe messaging. To achieve high performance despite the communication overhead of data updates, Twister uses relatively large-grained stateful map tasks. This heuristic is confirmed by the experiences from implementing applications for in-memory MapReduce. Furthermore, Twister inserts a local combine operation before the global reduce operation to determine whether to continue iteration. Using the in-memory framework, applications control the number of iterations to run. Consequently, in-memory MapReduce has a simpler API than Twister, and at the same time it gives applications better control over how data is handled.

MapReduce Online modifies the original MapReduce model to allow for online aggregation and continuous queries [50]. In addition to storing intermediate data temporarily on disk, HOP pipelines data directly between computing nodes, effectively converting MapReduce's original data-centric model to a message-oriented model. MapReduce Online broadens the field of application of the original MapRe-

duce model and increases the degree of parallelism even for legacy workload. On the downside, MapReduce Online fundamentally changes the MapReduce design. The HOP implementation involves buffer management, RPC-based communication and progress monitoring. On the one hand, our data-centric approach does not need explicit messaging. On the other hand, the performance and fault tolerance of in-memory MapReduce cannot profit from hand-tailored communication. However, the implementation of the in-memory MapReduce framework is much simpler and thus less error-prone than the implementation of HOP. As motivated above, the in-memory framework sticks to the original data-centric approach of MapReduce instead of requiring additional communication facilities.

To investigate and improve storage performance for one-pass MapReduce, Nicolae et al. have developed the BlobSeer storage as a replacement for the commonly used HDFS data store [144]. Like in-memory MapReduce, BlobSeer uses version-based concurrently control to handle object updates, which enables it to outperform HDFS under micro-benchmarks and real MapReduce workload. However, the objects used by BlobSeer are much larger than the programming language objects that in-memory MapReduce focuses on, such that BlobSeer does not rely on in-memory storage only. While in-memory MapReduce handles consistency mostly transparently for applications, BlobSeer makes versions explicitly accessible for applications. Despite the differences in purpose and design, BlobSeer makes a case that scalable consistency management is key to high-performance MapReduce.

The in-memory MapReduce framework has some similarities with Google's Percolator project [148], which enables incremental processing of large data sets as a replacement for MapReduce. In contrast to the in-memory MapReduce framework, Percolator is integrated in Google's closed-source infrastructure, entailing Google File System and the tabular data store Bigtable. Perlocator's transactions scale impressively well, almost linearly for more than five thousand cores. However, transactions cannot embrace more than a single row or column in Bigtable.

To enable modifications to trigger transaction execution, Percolator's designers have defined a notification mechanism. Notifications are similar in functionality to database triggers in that they are executed after a data update, but the observer runs as a separate transaction. Notifications serve a purpose similar to the proposed conditions, which also continue execution after a transaction has updated certain data. While Perlocator's observers randomly scan the table for pending notifications, the in-memory condition mechanism is driven by commit messages.

There exist several MapReduce implementations using data-centric communication, most prominently Phoenix [155, 195], Phoenix++ [180] and Ostrich [46]. However, unlike the framework presented in this chapter, these implementations target multicore processors with shared memory access. Ostrich optimizes Phoenix with respect to memory reuse, data locality and overlapping map and reduce phases. Phoenix++ improves the flexibility of Phoenix by using C++ templates, in contrast to in-memory MapReduce's approach of sensible default values, which the application can overwrite as needed.

6.5 Summary

The MapReduce programming model was primarily designed for single-pass computations, such that it cannot handle intrinsic data dependencies. Existing MapReduce frameworks for iterative and online processing handle data dependencies using additional communication facilities, which enlarges the framework design and complicates the programming model. In-memory storage helps MapReduce frameworks support data updates without any additional facilities. A transactional storage service can guarantee stronger consistency than the append-at-least-once semantics provided by the storage services used by legacy MapReduce frameworks. By using flexible consistency control, an in-memory framework can on the one side benefit from strong consistency where necessary for correctness and on the other side optimize for low-latency accesses by relaxing consistency. A key feature for the implementation of an in-memory framework for extended MapReduce are notifications about object updates, which replace the customized message-based communication facilities used by other implementations. Thereby, in-memory MapReduce allows a convenient and straightforward implementation of applications and downsizes the software package dependencies of the framework. Besides, the framework itself can build upon the notification mechanism to represent jobs and worker nodes in distributed

storage. Diverse use cases for in-memory MapReduce have been described in the chapter. These descriptions have focused on the access patterns induced by the applications as well as on the consistency requirements for iterative and on-line job execution. This chapter is based on two publications written by the author of this thesis [161, 162]. Figures 6.7 and 6.8 also appear in another publication by the author [158].

7

A distributed in-memory filesystem

The distributed storage service described in the previous chapters provides applications with shared objects. Therefore, the MapReduce applications described in the previous chapter communicate with the storage service through an object-based interface. A library-based implementation allows the storage service to replicate objects directly into the address space of the application. However, in case of legacy applications, developers often prefer not to modify the application. As an alternative to an object-based interface, filesystems offer a well-established set of functions to operate on files that are accessible to all applications on a computer. Distributed filesystems allow multiple computers to share the same file namespace.

This chapter describes a distributed filesystem that bases on transactional in-memory storage. After a motivation of distributed filesystems in the first section, the second section describes a user-level implementation that allows the filesystem to integrate seamlessly into the operating system interface. The third section presents two interface extensions to improve the scalability of slightly customized applications as well as an atomic append operation that builds on transactional semantics. The forth section discusses metadata structures that are optimized for concurrent file creation and access.

7.1 Distributed filesystems

The rise of cloud computing within the recent years has boosted the interest in large-scale computing environments. These environments come with high data storage needs. Distributed filesystems are a widely-used storage abstraction for cloud computing environments, providing scalable data management with a unified, well-known filesystem interface. Distributed filesystems are able to manage large amounts of storage. Compared to database management systems, filesystems leave management of data interrelations to applications. Nonetheless, the file abstraction is simple to understand and popular among programmers, such that filesystems remain an important paradigm for large-scale storage in the cloud era. For example, Google filesystem (GFS) is a distributed filesystem for large storage clusters [88]. Among other areas of application, GFS stores data for Google's highly scalable MapReduce framework. Google keeps the GFS closed-source, but the Hadoop project provides HDFS, a GFS-compatible open-source implementation [34]. To achieve high scalability in the number of nodes, GFS and HDFS abandon traditional POSIX compatibility in favor of special optimizations. For example, GFS has an append-at-least-once operation, which trades precision of operations in for a higher degree of concurrency [88]. HDFS currently implements write-once consistency only, such that files are im-

mutable after the creator of a file has closed it. The Hadoop developers are working towards support for modifying and appending operations [34]. This section presents the design and implementation of Elastic Filesystem (EFS), a filesystem based on transactional in-memory storage.

7.2 In-memory filesystem architecture

The EFS filesystem keeps its data and metadata in the Elastic Cooperative Random-Access Memory (ECRAM). ECRAM is a distributed in-memory storage service that supports adaptive consistency (see Chapter 3). It stores data in unstructured binary memory blocks, such that the filesystem needs to impose a structure on these objects. More specifically, the filesystem must implement file block management and a nameservice for arranging files recursively in directories. To enable efficient directory and file operations, EFS organizes the filesystem block allocation metadata and nameservice in B+-trees [49]. EFS creates a filesystem interface that is mountable under GNU/Linux operating systems using FUSE [66].

7.2.1 A B+-tree structure for directories and files

Like most other filesystems, EFS uses fixed-size blocks as a basic building unit for files. Each storage block provided by the service has a globally unique identifier. To arrange these identifiers into directories and files, EFS uses the B+-tree dynamic data structure. There are two different kinds of B+-trees in EFS: Directory trees for the nameservice and file trees for associating offsets within files with storage blocks. A B-tree is a balanced, ordered tree with multiple key/value pairs per node. The number of entries per node is called the order or branching factor of a B-tree. A B+-tree is a B-tree that stores all records in leaf nodes [49]. Multi-entry nodes benefit spatial locality, making B+-trees a popular choice for block-based filesystems such as Linux' upcoming Btrfs [65]. The balanced structure of B-trees guarantees insert, delete and search operations to proceed in logarithmic time, because the depth of the B+-tree is bounded by the logarithm of the number of nodes in the tree. Keeping all records at the leaf level allows efficient in-order traversal in a B+-tree, especially when linking leaves in a sibling list [49]. Directory trees associate filenames with file content. Each directory entry is indexed by a hash value of the corresponding filename and points to the file's metadata. EFS uses the CRC32 hash function, which can be calculated efficiently and distributes well enough. For file access patterns where preserving locality of filenames is more efficient, for example where each node accesses lexically proximate files, continuous hash functions could be used. File trees map offsets of blocks within the file to block content. In a file tree, each entry is indexed by the block offset and links to the storage for the specified block. Figure 7.1 exemplifies a directory tree consisting of a two-level directory B+-tree and a three-level file block B+-tree. Section 7.4 presents alternative metadata structures that favor scalability of file creation and access at the cost of reduced performance of directory listing.

7.2.2 In-memory storage for filesystem metadata

Distributed storage in EFS is provided by ECRAM, a flexible platform for in-memory objects. ECRAM instantiates a distributed storage service as described in Chapter 2. In the configuration used by EFS, objects are fixed-size memory blocks having a minimum size of 4 KB, and object IDs are 64 bits wide. Nodes reserve storage in large partitions, such that allocating an individual block does not require inter-node communication in the common case. ECRAM supports access-dependent consistency. For strong consistency, accesses are bundled in transactions, which are executed with ACID semantics (see Chapter 3). Durability is achieved by replicating modifications. To guarantee durability in case of severe failures, ECRAM could periodically write checkpoints to permanent storage. However, for non-critical purposes it suffices to create three replicas, similarly to HDFS's default replication value of 3 [34]. In some cases programmers can preclude transaction conflicts, so that accesses contend with weak consistency. Therefore, transaction validation is optional. Accesses outside of transactions are allowed, but they are not serialized with respect to concurrent accesses in transactional context. Non-transactional

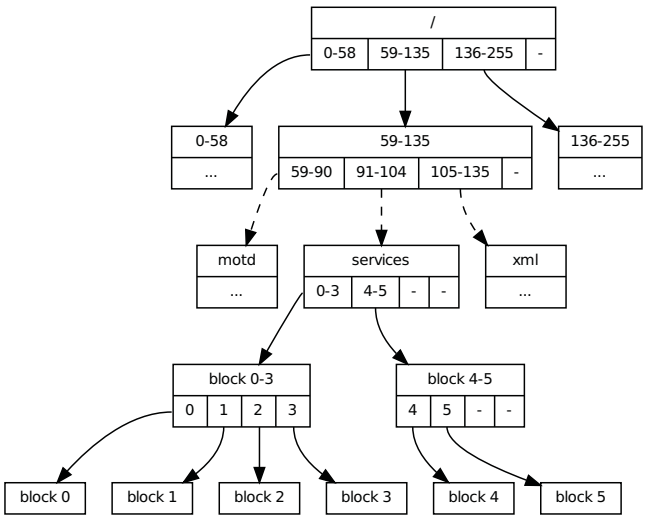


Figure 7.1: Nameservice and file blocks stored using B+-trees

accesses return valid but potentially outdated data. In order to maintain metadata consistency, EFS uses ECRAM's transactions over multiversion storage. Whenever EFS bundles a sequence of read and write operations into a transaction, ECRAM executes the accesses speculatively. When EFS reaches the end of the transaction, ECRAM validates the transaction against all other transactions that have occurred since the start of a transaction. If the validating transaction has not read or written an object that has been invalidated by a concurrent transaction, the validating transaction commits its modifications, else ECRAM rolls back the speculative changes and restarts the transaction from its beginning.

7.2.3 Cache synchronization

The implementation of the filesystem using FUSE simplifies the development, because filesystem crashes in user space do not affect the stability of the operating system kernel. However, the address space hosting the filesystem executable is separated from the address spaces of applications accessing files on the filesystem. Therefore, caching is done by three components of the storage system: the applications, the user-level FUSE service and the kernel buffer cache. Difficulties can arise from keeping these caches consistent. In case that the storage service updates replicas transparently in the address space of the FUSE service, it can effectively circumvent the buffer cache. In the example of Figure 7.2, buffer cache and application assume that the object contains value "A", while the storage service has updated the object to value "B". If the update happens transparently, the filesystem cannot invalidate the buffer cache. The FUSE interface defines a direct-I/O mode that makes file operations bypass the kernel buffer cache. If direct-I/O is selected, the kernel transfers data directly between FUSE service and application. However, if the filesystem is mounted in direct-I/O mode, applications cannot map the content of files into their address space using the `mmap` system call. There are two solutions to guarantee synchronized caches when memory mapping files. First, the filesystem can be implemented in the kernel, where it has direct control over the buffer cache. However, doing so requires a high engineering effort, because the programming environment in user level is more convenient than in the kernel. Second, the implementation of the storage service can avoid transparent updates to accessible objects by keeping accessible copies apart from storage-internal replicas. The separation of accessible and immutable replicas has been described in Subsection 3.1.4.

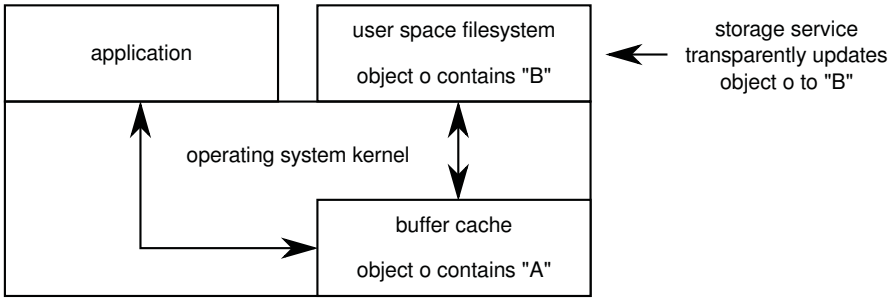


Figure 7.2: Cache synchronization issue with a user-level filesystem

7.2.4 Implementation of nameservice and file block management

The B+-tree implementation in EFS uses ECRAM's multiversion transactions to achieve atomic tree updates. In contrast to lock-free B+-tree implementations, using multiversion storage results in less write operations on the tree, which in turn reduces the amount of inter-node communication and benefits scalability (see Chapter 3). Transactions also allow atomic sequences of operations, for example when moving a file between directories. By means of speculative execution, DTM can perform even better than distributed lock-based synchronization if the contention on shared storage is relatively low. The transaction conflict rate describes the effective degree of parallelism in a DTM (see Chapter 5). In EFS, the conflict unit size equals the size of one B+-tree node, so that transactions conflict if they contend for the same tree node. Further conflicts can occur when accessing the same file structure. Read accesses in absence of concurrent modifications never cause conflicts. A B+-tree node in EFS stores pointers to its children, the keys that determine the value ranges of its children, a pointer to its next sibling and some bookkeeping information. Internal nodes have a maximum order of 339 entries, such that a node fits into an ECRAM block. Leaf nodes point to inode-like file structures that represent child directories or files. To establish a starting point for lookup operations, the B+-tree of the root directory is registered with a well-known object ID. The file structure aggregates the filename, the root of the attached B+-tree, the file size and further attributes such as access mode. The filename is needed for two purposes: first, during the `readdir` operation, to list directory entries, and second to distinguish different files with the same hash value of filenames. As detailed above, the B+-tree contains references to the data blocks of regular files and the entries of directories. Lookup operations in the EFS nameservice start at the filesystem's root B+-tree ("/" in the example of Figure 7.1), looking up the file structure for the top-level path component. The succeeding path components translate into further file structures, until the final file structure is reached ("services"). For accesses within the file, the block offset is calculated and then the respective blocks are looked up in the file's B+-tree (with 4-KB blocks, offset 8400 is located in block 2).

7.2.5 Filesystem interface

Users mount distributed EFS storage as a regular filesystem. The Filesystem in User Space (FUSE) kernel module forwards calls to Linux' filesystem API to the user-level EFS program [66], an approach first proposed by Eggert [73]. FUSE passes `mkdir` system calls to the handler function registered by EFS. Figure 7.3) shows the high-level code for creating directories. EFS runs the operation in a single transaction to avoid race conditions when several operations on shared data structures run concurrently. After asserting that the file does not yet exist, EFS creates a B+-tree node for the filenames in the directory. Afterwards, it allocates an inode that refers to the node and inserts it into the nameservice. The first marker of a transaction end will not cause a write commit, because the transaction has not yet changed any shared data. If the nameservice is locality-friendly, the transaction will frequently result in a local commit.

For `rmdir`, EFS first ensures that the directory exists and is empty, then it unsets the directory's

```
ecram_bot();
if (NULL != nameservice_get(path))
{
    ecram_eot();
    return -EEXIST;
}
btree_node *root = make_leaf();
efs_inode *node = make_inode(name, root, mode|S_IFDIR);
nameservice_set(path, node);
ecram_eot();
return 0;
```

Figure 7.3: Creation of directories

name in the nameservice and destroys the associated B+-tree. The `create` and `unlink` operations proceed similarly on file B+-trees. The `read` and `write` operations both start by looking up the file and the storage blocks corresponding to the specified offset. The write operation may require allocating new data blocks when appending to the file or filling holes in the file. Finally, ECRAM copies data between the input buffer or output buffer and the file blocks. In contrast to wide-area filesystems such as XtreemFS [104] and OceanStore [120], EFS assumes to run in a protected environment inside a single computing center. Therefore, it does neither entail access authorization nor data encryption. Adding these features to support federated clouds would be straightforward.

7.3 Optimizations

A user-level filesystem can implement optimizations that increase the scalability of adapted applications but at the same time retain the compatibility with unmodified applications. This section describes two such optimizations: first an adaptive mechanism that rearranges the underlying data structure depending on access characteristics, and second an atomic append operation comparable to GFS's append-at-least-once. In contrast to GFS, the append semantics neither produce undefined nor duplicate file regions. These evaluation of these enhancements is described in Chapter 8.

7.3.1 Adaptive tree balancing

The scalability of a distributed filesystem depends on the degree of parallelization enabled by the architecture. The runtime performance is furthermore determined by the actual file usage pattern. For example, if several nodes contend for creating files in the same directory, the filesystem must serialize accesses to shared data. On transaction-based storage such as ECRAM, concurrent accesses result in conflicting transactions. High conflict rates can slow down the filesystem to take even longer than serial execution. Although high conflict rates are not the common case, avoiding conflicts is desirable. The B+-tree structure in EFS lends itself to adaptive operation. A B+-tree node groups several entries into a conflict unit size. An insertion or deletion operation usually modifies only one or several entries in a node, it rarely modifies all entries in a node. However, a DTM always invalidates a conflict unit as a whole, causing false conflicts for unmodified entries (see Chapter 5). Reducing the number of entries per node avoids conflicts for subsequent tree modifications. The adaptive mechanism to avoid false conflicts in B+-trees consists of transaction monitoring and adaptive tree balancing.

Transaction monitoring

In order to derive information about false sharing in B+-trees, EFS associates the operations it executes with ECRAM's transaction monitoring information. ECRAM collects a number of metrics such as the

number of read and written objects per transaction and the transaction conflict rate. In general, it is difficult to distinguish false sharing from true sharing, because false sharing occurs with objects smaller than the conflict unit size. The problem of detecting false sharing bears some analogy to observing structures smaller than the resolution of a microscope. Although the transaction conflict rate does not directly identify false sharing, a high conflict rate indicates that data sharing exists and should be eliminated if possible (see Chapter 5). To eliminate false sharing, conflicting objects must be moved to separate conflict units. In case of true conflicts, the sharing situation will persist. Having separate conflict units does not harm, despite of the storage overhead. EFS stores the transaction conflict rate for each B+-tree. After having finished a transaction, EFS updates the conflict rate for the B+-trees accessed during the transaction. Given that different filesystem clients may have individual access patterns, monitoring data is stored locally and not transferred to other clients.

Splitting and aggregation of B+-tree

The B+-tree implementation supports dynamic adaptation of a tree's order. After having identified a B+-tree that is related to frequent transaction aborts, EFS reduces the default order of the tree. There is no need to immediately restructure the tree, because subsequent insertions and deletions of items will eventually modify the tree anyways. Until then, queries benefit from the previous larger order. In case the tree causes relatively few transaction aborts or is rarely accessed at all, EFS could increase its order. However, in most cases ECRAM's storage capacity outweighs the space saved by coalescing nodes.

7.3.2 Flexible consistency management

Many cloud computing applications generate large amounts of data, but do not modify data once written. For example, the MapReduce model accesses data with write-once-read-many consistency in both the map phase and the reduce phase.

Append-at-least-once

To efficiently support the write-once access pattern, GFS offers an append-at-least-once operation. This special operation guarantees that a data block is written at least once, such that subsequent read operations on different nodes consistently see the data. The append-at-least-once operation may result in the data block being appended more than once. Only one data block is guaranteed to be defined on all nodes, the other blocks are considered undefined by GFS [88]. Tolerating writes that occur more than once benefit scalability. If different append operations interleave, only one of them succeeds, and the others are re-executed transparently to the application. However, GFS leaves it to the applications to detect and ignore undefined regions or duplicates in files, for example by calculating checksums and filtering unique application-defined IDs. Ghemawat, Gobioff and Leung argue that duplicates occur rarely, such that many applications can ignore them [88]. Use case analysis shows that there exist many applications which cannot tolerate duplicate data. For example, inaccuracy aggravates if results are computed iteratively or if intermediate results serve as input for another computation. Therefore, the DTM-based append operation never creates duplicates. A GFS-like append-at-least-once operation could be implemented in EFS as follows: The filesystem reads the current length of the file to append to. Then it writes the supplied data to the offset corresponding to the end-of-file value read before. If the new file length matches the expected end-of-file value, it returns from the call, otherwise it repeats the operation.

Implementation and optimization of atomic append

Taking advantage of the ECRAM DTM, EFS implements an atomic append operation similar to the append-at-least-once operation described above. EFS executes the append operation as a single transaction. In terms of append-at-least-once described above, reading the file length, appending data to the file and checking the new file length becomes an atomic action. Conflicts between concurrent append or write

```
file_blocks *blocks  = prepare_blocks(file, data, offset, length)
ecram_bot();
ecram_inode *inode = nameservice_get(path);
if (NULL == inode)
{
    ecram_eot();
    return -ENOENT;
}
append_blocks(inode, blocks);
length += pad(inode->i_size);
inode->i_size += length;
ecram_eot(0);
return length;
```

Figure 7.4: Atomic append operation

operations are detected by ECRAM, and the transactions having read outdated content are restarted automatically. Transactions should be kept as short as possible to avoid unnecessary conflicts. Considering that the data to append is private until the append transaction finally commits, the data blocks can be written before starting the append transaction using weak consistency. This optimization only requires padding the file to reach a length that is a multiple of the data block size. Figure 7.4 shows the simplified code for the atomic append operation.

7.4 Metadata management using a hashtable and partitioned directories

The previous sections have discussed filesystem metadata management based on B+-tree structures. As discussed in Section 7.2, B+-trees favor caching using a fan-out that matches the block size of the underlying flat storage. However, in some distributed applications, many nodes access files in the same directory at the same time. A filesystem can avoid false sharing situations for this kind of applications by randomizing metadata placement using hashing. As an alternative to B+-tree based metadata management, this section presents a metadata structure that favors parallel file creation and path lookups using partitioned directories and an in-memory hashtable.

7.4.1 Partitioned directory tables

The two basic operations on filesystem metadata are accessing files and listing directory entries. In many applications, file accesses such as creation, deletion and data transfer occur frequently, whereas directories are seldom listed. Therefore, a filesystem should optimize the performance of the former operation.

A distributed filesystem optimized for file manipulation should modify only local data structures during file accesses, such that conflicts with concurrent operations are avoided by design. With this locality optimization, the metadata for files in the same directory can be dispersed over many storage nodes. Thus, listing a directory requires fetching metadata from different nodes. Figure 7.5 displays the resulting metadata structure. Each storage nodes that creates files in a certain directory keeps a local table of entries in the corresponding directory, called a direntry table. The direntry table can be regarded as a partition of the complete directory. It is placed in shared storage such that it can be read by any node, but only one node writes to it. A global table for the directory references all existing local tables.

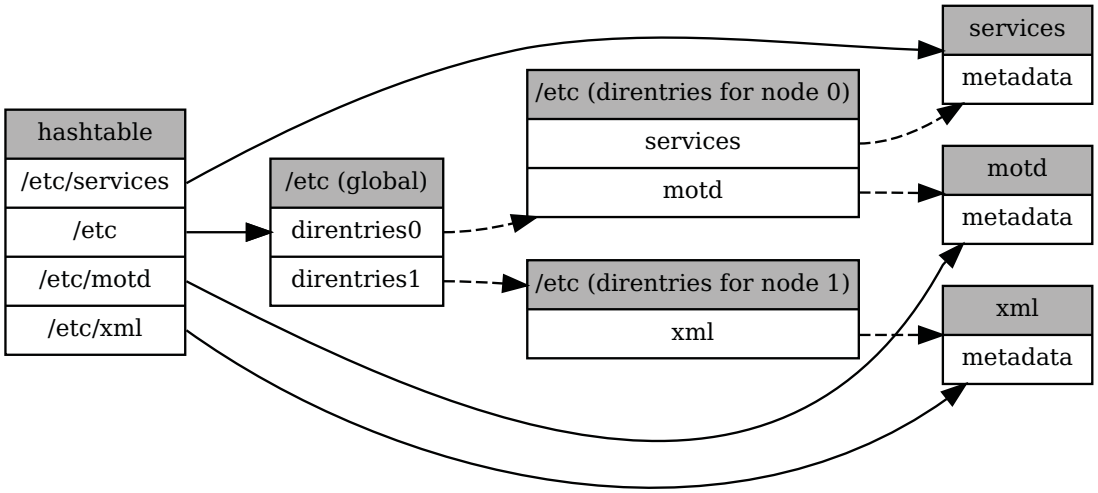


Figure 7.5: Architecture of a hash-based filesystem

In order to access file metadata efficiently, the filesystem can hash full pathnames to metadata entries. Access to file metadata does not traverse the directory listings of the leading components of its pathname. The hash-based lookup has an O(1) runtime complexity. By combining hash-based file lookup and partitioned directories, the filesystem retains hierarchical structuring of the namespace and improves the scalability of concurrent accesses to the same directory

Some operations on files, such as unlinking a file, access the metadata corresponding to a filename. An obvious solution requires at least three storage accesses, namely the hashtable entry for the directory holding the metadata, the global directory and successively the per-node direntries. In the worst case, the lookup traverses all direntries for the directory until it finds the metadata entry for the file. Even the average number of storage accesses for reverse lookups is linear in the number of nodes, which results in bad scalability. The iterative lookup can be improved by introducing backlinks from a file to the respective direntry table and from the direntry table to the global directory. If backlinks are available, the filesystem can directly lookup the filename in the hashtable and retrieve a reference to the direntry from the file's metadata, which results in only two storage accesses, independent of the number of storage nodes.

When creating a file, a storage node first needs to check whether the file already exists. Therefore, it looks up the pathname in the hashtable. If the lookup is successful, the flags passed to the creation call determine whether the creation fails or whether it returns the existing file. A failing lookup means that the file does not exist yet. In this case, it also looks up the pathname of the directory in which the file is created. If the directory does not exist, the creation fails. However if the directory exists but a local direntry is not there yet, the direntry is created and inserted into the directory. Finally, the node creates a new metadata entry and links it to the direntry table. In the common case that a direntry already exists, the described algorithm cannot cause any conflicts. Conflicts can only occur in the rare case when inserting a direntry into a global directory.

The implementation of accessing a file to read or write data is straightforward. Metadata is retrieved using an O(1) lookup in the hashtable, and the read or write operation proceeds as detailed in Section 7.2. The deletion of a file requires retrieving not only the file's metadata, but also the corresponding direntry, in which the file is marked as deleted. The direntry access benefits from the backlink technique explained above. An empty direntry can be deleted, but keeping it for future insertions may also be sensible. If the file is deleted by another node than the node that created it, the operation modifies a remote direntry, a situation similar to a remote free (see Subsection 2.2.4) that can also access conflicts.

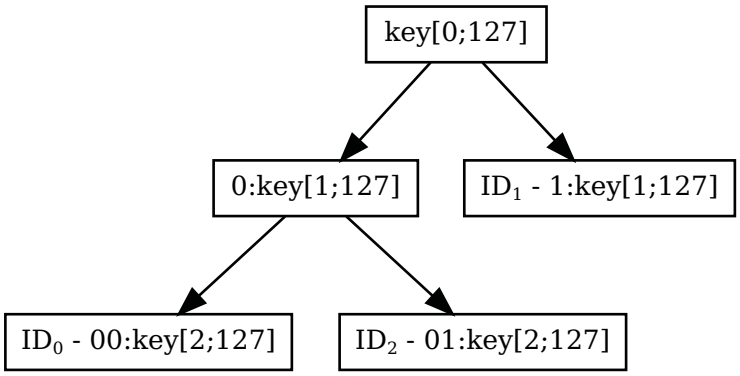


Figure 7.6: Keyspace partitioning with prefix matching for three nodes

7.4.2 A hashtable-based in-memory nameservice

The hashtable that stores nameservice entries can be implemented in a straightforward manner. In contrast to DHTs, which are described in Chapter 2, the in-memory hashtable makes all entries directly available for all participants, such that it benefits from caching. Certain assumptions about the environment enable specific optimizations. In a secure environment such as a private cloud, the hashtable implementation needs not consider malicious nodes and neither authentication. If node churn is low compared to the number of file operations, the hash mechanism can use a simple partitioning scheme that ensures an equal partitioning of the keyspace.

Under the described assumptions, the hash mechanism can store the total number of nodes and the hashtable in the shared storage, which allows to achieve an almost equal partitioning of the keyspace. Consequently, the total number of nodes is known, so that the nameservice can ensure that the nodes have contiguous IDs from 0 to the total number of nodes minus one. The relationship of nodes and nameservice entries can be based on prefix matching, which partitions the keyspace in the form of a binary search tree (see Figure 7.6). Concerning the hash function, the filesystem can use the MD4 algorithm, which is simple to implement and fast to compute. When joining the system, a node retrieves the current total number of nodes, initializes its ID and increments the number of nodes by one. Its new ID represents half the keyspace previously assigned to another node, so it contacts the node and changes the ownership of the corresponding hashtable entries. Node departure is handled symmetrically. When a node leaves the system, at most a single peer needs to take over its ID. The orphaned hashtable entries, for which the leaving node was responsible, are reassigned to the node whose ID matches the longest prefix of the hash values, and the total number of nodes is decremented by one.

Fault tolerance needs to address two potential issues. The first issue is loss or inconsistency of the global state of the system: the number of participating nodes and the routing table. By using transactions on replicated storage, the filesystem avoids this issue. When key duplicates are acceptable for a short period of time, quicker parallel updates of the global information could be possible during the joining process by using less-than-strict consistency for the transactions. In this case, contending nodes would eventually need to merge their data, and all but one of them would have to start a new joining process. Weak consistency could also be used when a node is leaving. Nodes might need to pull up-to-date global information when they are unable to contact some node in order to ensure that it is still part of the system. All cases of weak consistency handling however need further specification before they can be implemented.

The second potential issue is the loss of data due to node failure. However, if all information is stored in shared storage, the danger of losing data is handled by the replicating storage service. This allows nodes to give the object ID pointing to the root of their data to a few backup nodes, all the actual data replication being done by the storage service. When either the storage or an implementation-specific technique detect a failure, the backup nodes for the failed node determine which node will

temporarily take over the failed node's data. This node will then engage in the node departure protocol while assuming the identity of the failed node.

7.5 Related work

The body of work on distributed filesystems is huge. Among the mature distributed filesystems often used in production environments are NFS [172], AFS [102] and CODA [173]. These distributed filesystems have been designed for general-purpose file serving in LANs. Storing all name information and file data at central servers limits NFS's scalability. CODA and AFS improve on NFS by supporting replicated servers and fault-tolerance. POSIX-compliant implementations of these distributed file systems are available for many operating systems [183]. POSIX-compliant filesystems enforce close-to-open consistency, which is described by Satyanarayanan [173]: A file that is newly opened sees the result of the last close on that file. Close-to-open consistency implies that the close system call must carry out many bookkeeping tasks. Compared to these filesystems, EFS aims at adaptive operation, depending on imposed workload. The decentralized and transactionally consistent storage and meta-data enable better scalability than comparable centralized systems. At the same time, the consistency guarantee is stronger than close-to-open, in that it handles each storage operation as a transaction.

Some POSIX-compliant distributed filesystems target higher scalability and reliability than the previously mentioned systems. IBM's distributed zFS filesystem is based on a cooperative cache and distributed transactions [167]. The cooperative cache is a distributed data store, which avoids a centralized storage server much like ECRAM. The distributed transactions in zFS are more similar to database transactions than the DTM transactions in ECRAM. XtreemFS is a distributed and replicated filesystem that aims at wide-area setups in the Internet [104]. Like EFS, XtreemFS is mostly implemented in user space, and therefore it uses FUSE to attach to virtual filesystem of the kernel. To tolerate high communication latencies, XtreemFS supports file striping and read-only replication. As described in Chapter 4, the in-memory objects accessed by EFS are replicated on demand to each server mounting the filesystem. XtreemFS uses a dedicated metadata server running a special key-value database, whereas EFS stores all data and metadata in distributed storage.

The file access characteristics and scalability demands of large-scale data-processing applications prompt for distributed parallel fault-tolerant filesystems, which are highly scalable and resilient to server failures. The most popular distributed filesystems for MapReduce-like workload are Google filesystem (GFS) [88] and its open-source clone HDFS [34], which have been described above. While these systems also support integration into conventional operating system APIs, they come with additional features which necessitate special APIs. An example for such a feature is GFS's append-at-least-once operation described above. The EFS filesystem supports a comparable append operation which never generates duplicate regions, hence applications can rely on all file regions to be defined, without the need for further coherence checks. GFS guarantees atomicity of namespace modifications by having a central server, whereas EFS places directory structures in distributed memory and uses DTM to achieve atomic modifications. Several optimizations for B+-trees have been proposed to speed up operations. Rebalancing and linking have been suggested to reduce locking during accesses, for example sibling pointers [126]. Wang proposes storing B-trees in multiversion memory to increase performance and scaling in parallel systems [191]. The tree balancing algorithm applies to the similar case of B+-trees on versioned DTM, however it adaptively manages trees by taking into account the transaction performance.

7.6 Summary

Filesystems are a well-known abstraction for storage at the operating system level. This chapter has demonstrated that a distributed in-memory filesystem can be constructed on top of a storage service and thus provide an alternative interface to access in-memory storage. Beyond previous work on distributed filesystems, it has contributed an adaptive technique to reduce the number of conflicts on

B+-tree structured metadata. It has also contributed an atomic append operation, which is generally difficult to support for a distributed filesystem. A third contribution is a distributed hash-based metadata structure that allows highly scalable parallel operations on files even in the same directory. The contributions of this chapter have also been presented in a scientific publication [159]. The hash-based filesystem is described in a technical report [112].

8

Evaluation

The previous chapters have described the design and implementation of several components of in-memory storage systems. Chapters 2 to 5 have discussed a storage service including support for variable consistency, smart replication and adaptive conflict granularity. Chapter 6 has described a framework for extended MapReduce computations as well as a number of applications using the framework. Finally, Chapter 7 has proposed a distributed filesystem that benefits from running atop the in-memory storage service. The discussion of storage service, filesystem, MapReduce framework and applications has focused on their performance and scalability. This chapter emphasizes these non-functional properties and presents measurements to support the claims. The *Elastic Cooperative Random-Access Memory* (ECRAM) project implements an in-memory storage service that includes many contributions of this thesis. Therefore, the measurements were conducted using ECRAM.

8.1 Implementation effort using in-memory storage

In-memory storage of application data not only reduces access latency compared to storage on disk, but it also simplifies implementation. Neither operating system kernel nor application code need to care about relocating data from RAM to disk. However, a storage system without disks has certain requirements. First, the payload data must be replicated in order to achieve sufficient reliability. Second, the capacity of the system must be large enough to hold all data including metadata. Third, since data cannot be swapped to disk, it must be distributed among the participants of the system. The first and third requirement have been discussed in Chapter 4. The requirement of sufficient capacity depends on the configuration of the system and on the imposed workload. Given that the sizing of hard disks poses similar questions, it can be concluded that storage management becomes rather simpler than more complicated with an in-memory architecture.

The manifestation of an in-memory storage system as a user-level library further simplifies its implementation. Chapters 3, 4 and 5 describe several cases where the library implementation allows the storage detailed insights on application semantics, for example to implement transparent restart of memory transactions and to define object access groups. These insights cannot be assessed easily when the storage service is located in a separate address space.

The effort of programming on transactional memory (TM) has been discussed in literature. In a comparison between locking and TM, McKenney et al. observe that TM simplifies concurrent programming [135]. However, the authors also discuss several weaknesses of TM. In case of high conflict rates,

component	SLOC	comment
interface	853	
base	493	
net	2286	most code for epoll-based TCP networking
consistency	1915	
replication	1730	
monitor	950	
object	6686	including 3249 SLOC for a port of Dlmalloc
lib	11884	including 9651 SLOC for a port of Glib
debug	543	
apps	18314	some applications ported from STAMP [39]

Table 8.1: Source lines of code (SLOC) of the ECRAM in-memory storage system

TM scales worse than equivalent lock-based synchronization, whereas in case of few conflicts, transaction execution and validation compromise TM's performance. The adaptive conflict granularity approach presented in Chapter 5 of this thesis enables a TM to accommodate to varying conflict rates. McKenney et al. also discuss restrictions of non-restartable I/O and poor hardware integration. They conclude that many programmers are more familiar with lock-based synchronization techniques, such that the latter will prevail for several years. Pankratius and Adl-Tabatabai compared the programming skills of graduate students using TM or locking [147]. The authors report that the student teams using TM progressed faster and produced code that is easier to understand. However, performance tuning was more time-consuming with TM than with locking.

DTM can serve as a practical example when teaching parallel and distributed programming. ECRAM has been used in a course on distributed systems at Universität Düsseldorf. The lecture introduced the theory of TM and the MapReduce model and also discussed an example application for in-memory MapReduce. Furthermore, the students were provided with the manual printed in the appendix of this thesis. All students were able to setup the system and to start the example application. In the following practical phase, ECRAM was used to implement MapReduce applications. The students discovered and analyzed several peculiarities of transactional programming, for example transparent restart of failed transactions and slow progress in case of high conflict rates. Advantageous for the understanding of the workings of the DTM was the simple library interface and the limited size of the applications. The oral examinations at the end of the term addressed TM, and most students passed the questions with only minor mistakes. This experience shows that practical programming on a DTM can successfully support parallel and distributed programming courses, especially to activate participants to reason about synchronization issues.

Despite its flexibility in usage, the implementation of the distributed in-memory store ECRAM measures only a few thousand lines of code. The numbers of source lines of code presented in Table 8.1 were generated using David A. Wheeler's SLOCCount program. The implementation of the extended MapReduce framework consumes 1220 SLOC in the component *lib*, and the in-memory filesystem requires 1197 SLOC in the component *apps*.

8.2 Performance and scalability of in-memory storage

As already mentioned in Chapters 6 and 7, many aspects of in-memory storage discussed in this thesis have been implemented. The research storage system ECRAM builds for x86-64 compatible CPUs running the GNU/Linux operating system. After presenting the infrastructure used to measure the storage, this section evaluates the results of smart replication and adaptive conflict granularity. Furthermore, it analyzes the performance and scalability of in-memory MapReduce and the distributed in-memory filesystem.

8.2.1 Hardware used for measurements

The measurements were taken on two different testbeds. Testbed A is a cluster of 33 dual-core machines equipped with AMD Opteron CPUs. 17 machines have Opteron 246 CPUs operating at 2 GHz, and 16 machines have Opteron 244 CPUs operating at 1.8 GHz. Each machine is configured with 2 GB cache-coherent NUMA RAM. The machines in Testbed A boot Debian 6.0 Squeeze with a Linux 2.6.32 kernel via NFS, because they don't have any permanent storage such as harddisks. The cluster is connected using Gigabit Ethernet. All experiments have been repeated five times, and results are given as the arithmetic mean numbers, unless stated otherwise.

Testbed B consists of 8 machines equipped with AMD Opteron 4122 quadcore CPUs, which operate at 2.2 GHz. The machines have each 16 GB of RAM and are connected over Gigabit Ethernet. They use the same OS and boot configuration as Testbed A. The nodes in Cluster B generally have a lower ratio of clock cycles per instruction retired (CPI), which means that local computations are faster, but that the latency of network communication in both clusters is equal.

8.2.2 Smart replication

Chapter 4 has presented two conventional update propagation protocols, invalidate protocol and update protocol and proposed a smart replication protocol that switches dynamically between sending invalidations and updates. On the one hand, sending invalidations instead of updates can save network bandwidth and thus indirectly reduce communication latency. On the other hand, using the update protocol can reduce the latency of updating an invalidated replica.

Three applications (k-means, real-time raytracing and labyrinth) and a microbenchmark providing diverse access patterns serve to evaluate the smart replication policy. K-means and raytracing execute on top of the extended MapReduce framework of ECRAM (see Chapter 6). Furthermore, labyrinth and k-means are based on STAMP [39].

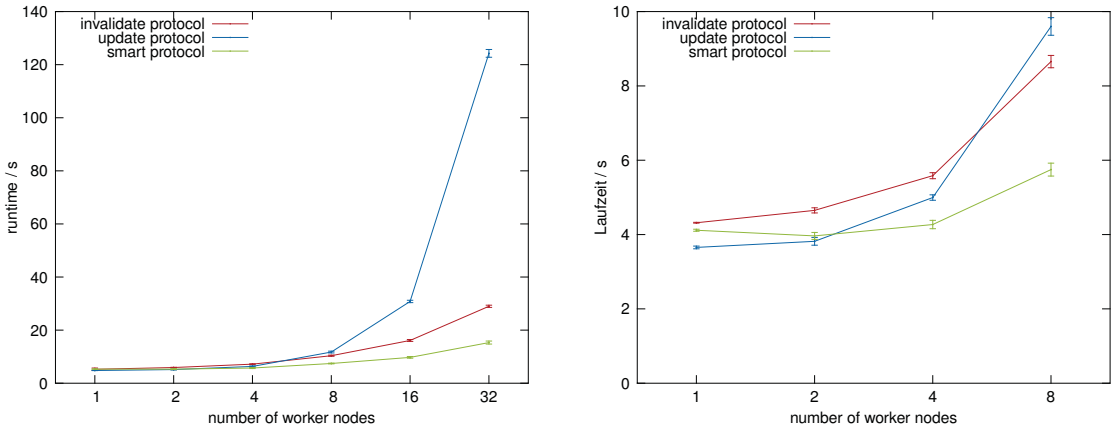
Raytracing

This application implements real-time raytracing for dynamic scenes including effects such as shadows and reflections. The scene used to conduct the measurements simply shows a bouncing sphere. More complex scenes would increase runtime without allowing new insights concerning the replication mechanisms.

The computation has been described in detail in Chapter 6. The master node allocates memory for one image frame and splits the image in n parts, one for each worker. After all workers have completed calculating their part, the master node copies the parts into local memory and displays the full frame. The application then adjusts the sphere's coordinates in order to simulate a bouncing movement. The workers calculate the next frame, and the iteration continues.

In order to demonstrate storage aggregation, the tests increase the image size linearly with the number of nodes. The image size in pixels P is calculated as $P = 1920 \cdot 1080 \cdot N$, where N is the number of worker nodes.

Worker nodes begin sending updates as soon as they have completed their calculation when using update-only or smart replication. However, update-only replication sends updates to all nodes, but only the master needs them, because the intermediate data is never read by subsequent Map phases.



(a) Cluster A (b) Cluster B

Figure 8.1: Execution time of raytracer (image size 1920x1080xN pixels)

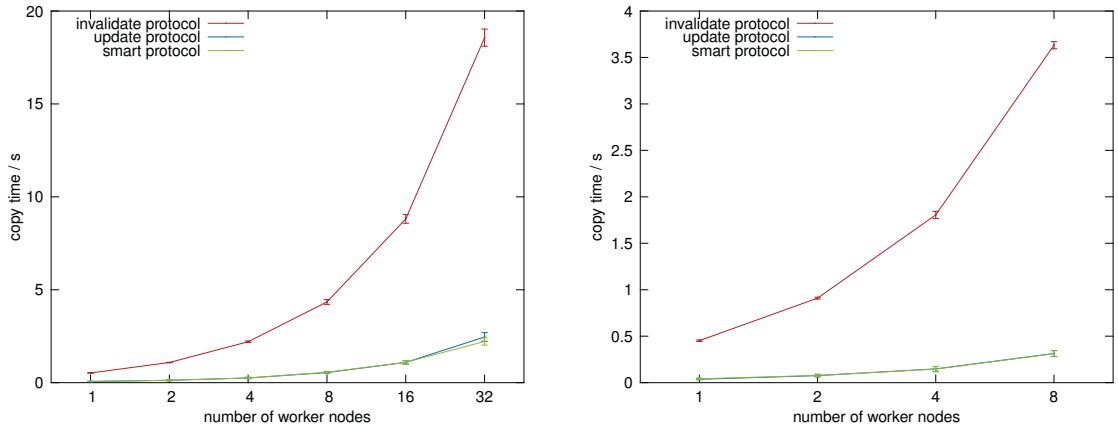
The master accesses many configuration and coordination objects for every frame but only once the image. Thus the evaluation would delete the image entries in all access monitors since the evaluation concentrates on the quantity of accesses. Therefore, when using smart replication, the application sets the replication hint to send updates always to the master.

Figure 8.1 shows that runtime increases with the number of nodes, because of the increasing image size. The standard deviation of the data is displayed as whiskers that extend above and below the mean value. Cluster B needs less time to raytrace the image, because it is equipped with more powerful CPUs, but the standard deviation of the overall runtime on both clusters is in the same range between 0.1 and 1 second. With 32 nodes, the update protocol fully uses the communication network. Smart replication outperforms the other replication approaches, especially for larger image sizes. Figure 8.2 displays the time the master needs to copy the parts of a frame to a local buffer. When using an invalidate-only replication approach, this is very time-consuming, because the storage accesses cause a step-by-step transfer of related objects from remote nodes. For update-only and smart replication, the server has all data already in place. The copy operation on Cluster B takes less time than on Cluster A, which can be attributed to the higher transfer date between network interface and RAM. Finally, Figure 8.3 shows sent data, which is (as expected) worst for update-only and almost the same for invalidate-only and smart replication. Invalidate-only replication sends the same data but also in addition the request packets. Transaction conflicts are not an issue here for any replication mechanism, because data is aligned to avoid conflicts and the scene graph is updated synchronously with respect to frame calculations.

Given that the master only requests those parts of the image that have changed with the relatively fine granularity of 4 KB, delta encoding is not helpful in this application. Sending updates along with validation requests to the master enables the latter to access the next image frame parts as soon as possible.

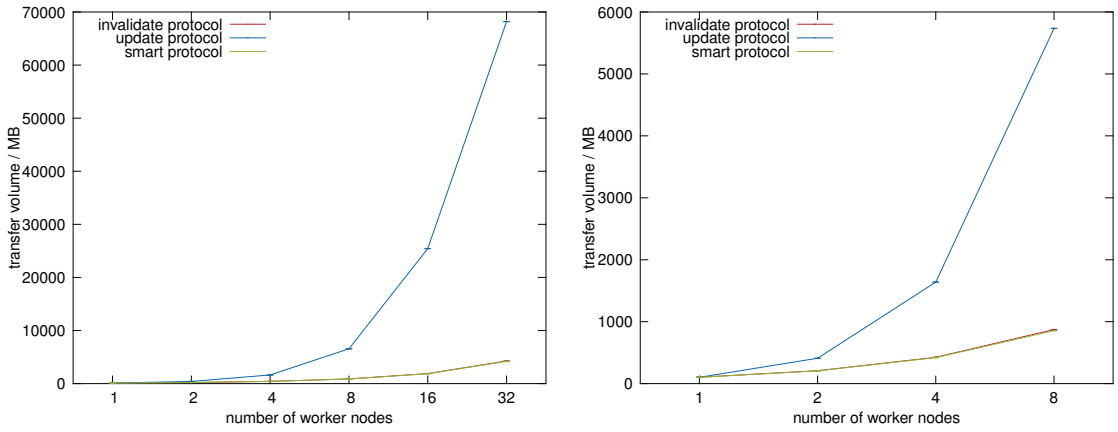
K-means

K-means is a clustering method used in data mining. It aims at partitioning n observations into k clusters, in which each observation belongs to the cluster with the nearest mean (cluster center). The iterative algorithm has been described in Chapter 6. The input file for the testbed A contains 1,000,000 points in a three dimensional space with 16 cluster centers. Because of insufficient storage capacity, the testbed B can be used with 500,000 input points, only. Update-only replication does not scale well, because it sends too much irrelevant data. All nodes receive all changes, but only a subset of the information is necessary. Smart replication reduces runtime, copy time and sent data as shown in Figure 8.4, Figure 8.5 and Figure 8.6. However, there is a phenomenon with increasing number of nodes. When us-



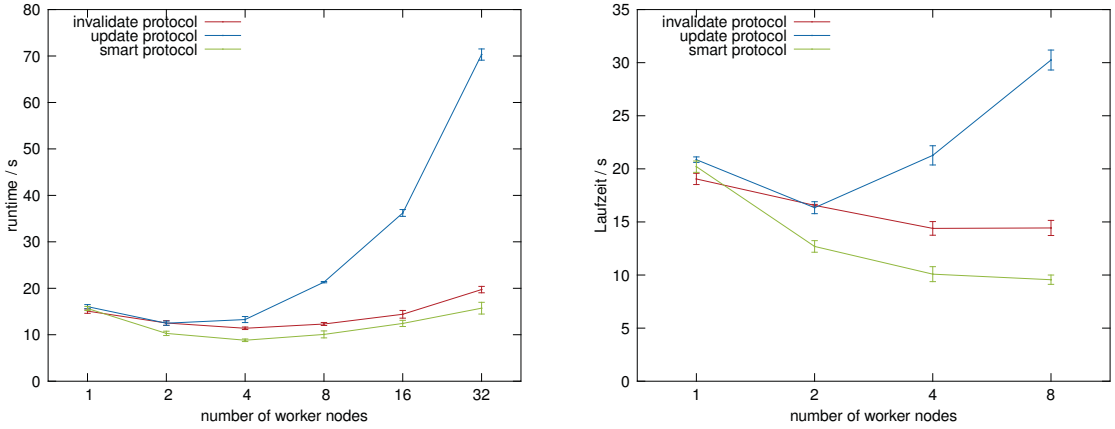
(a) Cluster A (b) Cluster B

Figure 8.2: Time for copying the generated image of raytracer (image size 1920x1080xN pixels)



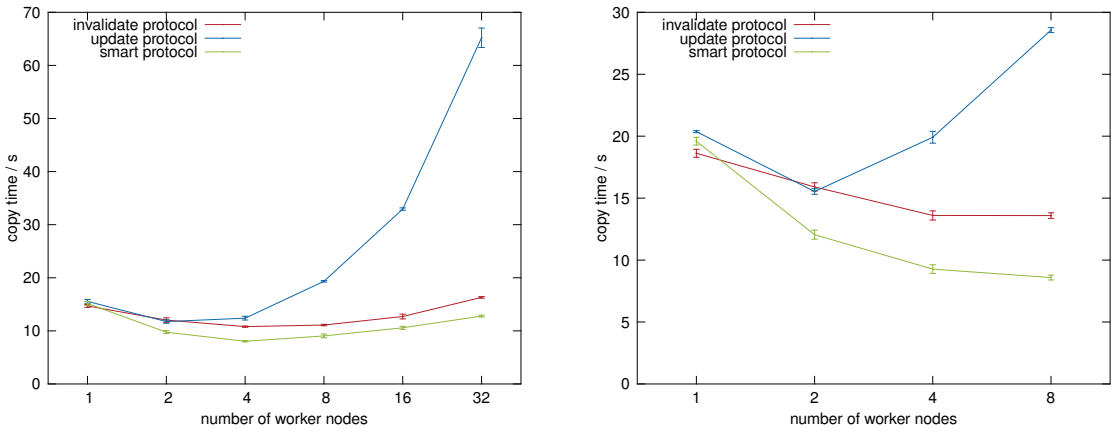
(a) Cluster A (b) Cluster B

Figure 8.3: Data volume of raytracer (image size 1920x1080xN pixels)



(a) Testbed A with 500,000 points (b) Testbed B with 1,000,000 points

Figure 8.4: Execution time of kmeans, 3 dimensions, 16 clusters



(a) Testbed A with 500,000 points (b) Testbed B with 1,000,000 points

Figure 8.5: Time for copying results of kmeans, 3 dimensions, 16 clusters

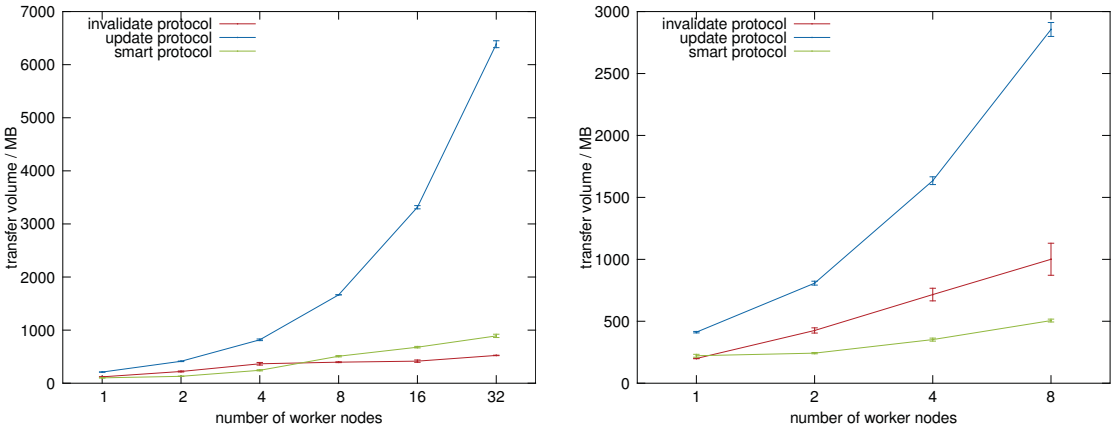
ing smart replication, every new replica of an object is distributed as long as it is accessed frequently by some node. If transactions are very short, some updates may be skipped, because they arrive too late. Thus not every replica is necessary and there is some overhead compared to invalidate-only replication.

Delta encoding reduces the amount of data sent, because only cluster center positions change. Sending updates with the validation requests to the master node is helpful, because the master needs all cluster center positions to calculate deltas.

Labyrinth

Labyrinth is an application for finding routes in a maze using the Lee algorithm with breadth-first search. The same algorithm is often used in electronic design automation. All nodes calculate paths in a shared grid for given source and destination points. The latter are read from an input file and stored as jobs in a shared queue. The measurements have used a grid size of 1024*1024*16 and 256 paths to be routed.

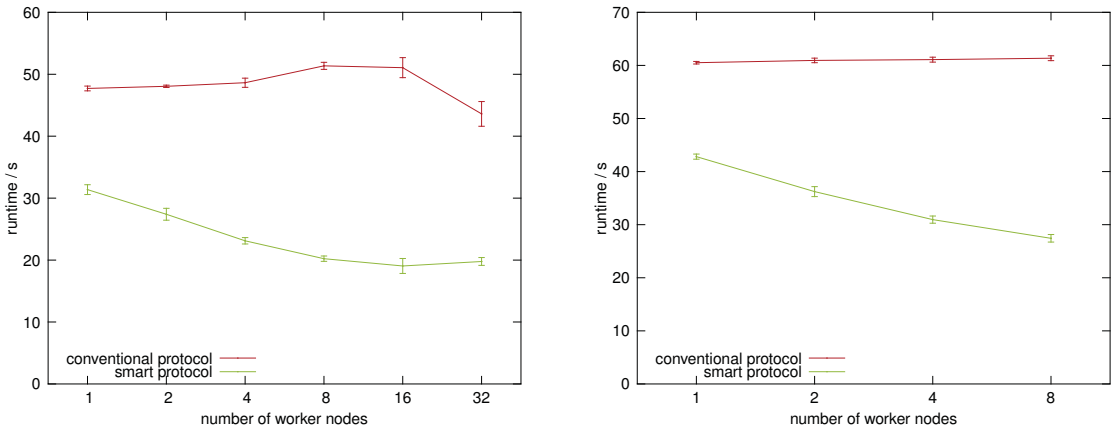
Labyrinth has a large read set during the wave expansion phase. The backtracking phase will then write back the route into the shared grid, which may cause conflicts with transactions that have routed



(a) Testbed A with 500,000 points

(b) Testbed B with 1,000,000 points

Figure 8.6: Data volume of kmeans, 3 dimensions, 16 clusters



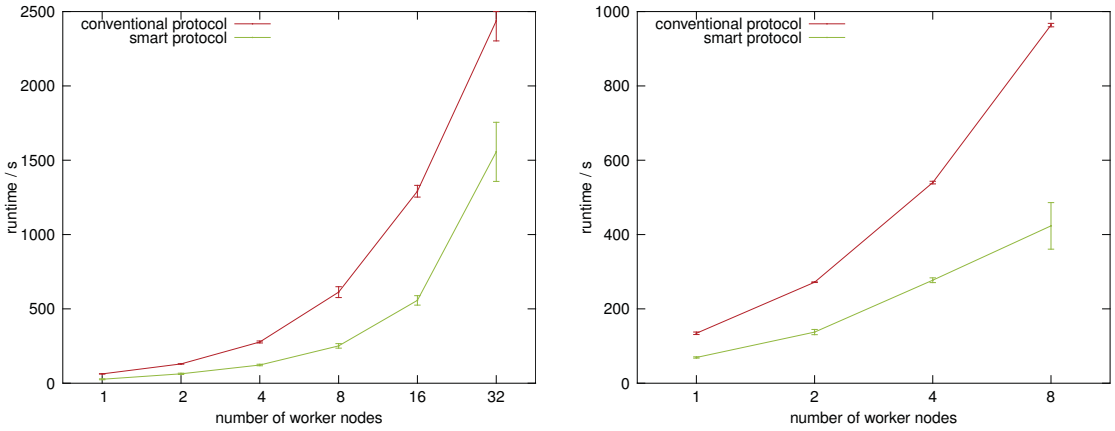
(a) Cluster A

(b) Cluster B

Figure 8.7: Execution time of labyrinth with 1048576 cells, 3 dimensions, 128 paths

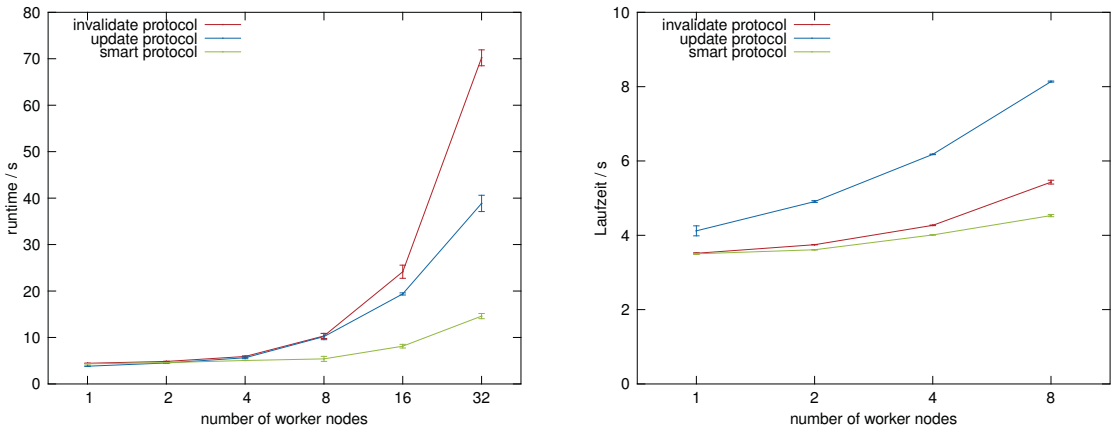
paths crossing the one to be written into the shared grid. In that case conflicting transactions will abort and restart from the beginning. Obviously, labyrinth is a non-trivial application, which has a considerable conflict probability when using transaction, see also LeeTM in [15]. Smart replication has introduced snapshot isolation for ECRAM which is not supported by invalidate-only replication.

Figures 8.7 and 8.8 show that labyrinth scales well with all input files when smart replication is used and larger grids scale even better, as expected because of the reduced conflict probability for routes. Invalidate-only replication has severe scalability problems because of its huge read set caused by the wave expansion. The latter is no problem when using smart replication because snapshot isolation avoids unnecessary transaction aborts caused by routes written by other transactions in the shared grid. But of course conflicts can still occur during the backtracking phase as described above. However, this is not always the case as shown by the measurement results. The dominating optimization introduced by smart replication for labyrinth is snapshot isolation. Delta encoding, distributed commit notifications etc. have no evidential impact on runtime and scalability.



(a) Cluster A (b) Cluster B

Figure 8.8: Number of conflicts for labyrinth with 1048576 cells, 3 dimensions, 128 paths



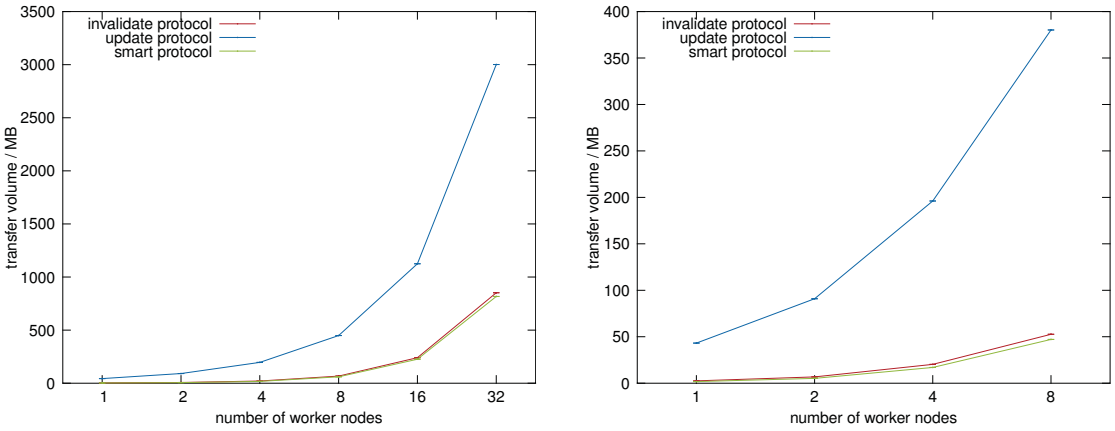
(a) Cluster A (b) Cluster B

Figure 8.9: Execution time of maxpar

MaxP microbenchmark

In order to evaluate the distributed commit notifications and the delta encoding, a dedicated microbenchmark for maximum parallelism (MaxP) has been implemented. The MaxP benchmark runs one loop with 10,000 iterations on each node where each iteration is executed as one transaction incrementing a counter variable. Variables are carefully allocated avoiding false transaction conflicts, which allows maximum parallelism. Nevertheless, MaxP requires a high transaction throughput, because all transactions are very short (shorter than typical transactions).

With update-only replication, every node receives all updates of all counter variables, although not needed. So smart replication outperforms as expected the update-only replication approach, see Figure 8.9. Distributed commit notifications used by smart replication relieve the master from sending out all commits like done for invalidate-only replication. The distributed sending explains why smart replication outperforms an invalidate-only approach (see Figure 8.9 and Figure 8.10).



(a) Cluster A (b) Cluster B

Figure 8.10: Data transferred for maxpar

8.2.3 Adaptive conflict granularity

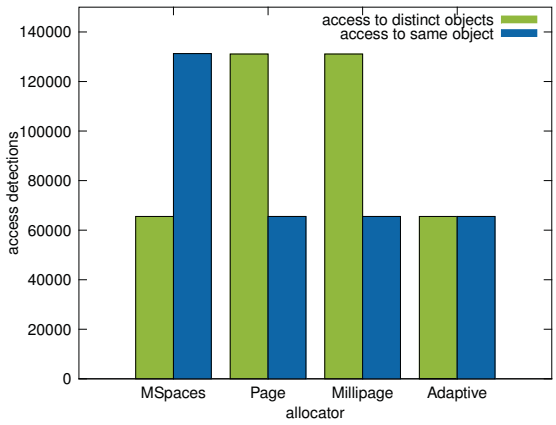
To evaluate the performance of the adaptive sharing technique introduced in Chapter 5, different sharing and allocation strategies have been compared using microbenchmarks. An on-line data processing application demonstrates that transactional memory benefits from adaptive sharing for realistic workloads. The experiments ran on two dual-core nodes equipped with AMD Opteron 244 processors running at 1.8 GHz from cluster A (see 8.2.1). The nodes were connected via Gigabit Ethernet over Broadcom NetXtreme NICs.

Microbenchmarks

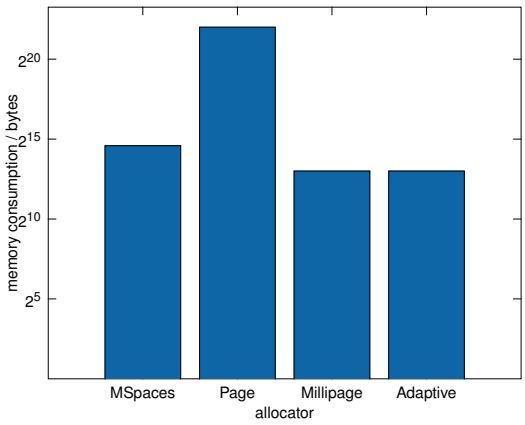
The synthetic workloads have been used with four different allocation schemes. The dlmalloc allocator is a general-purpose allocator, similar to the one used by the GNU standard C library. For the presented experiments, dlmalloc has been used in the Mspaces configuration, in which each thread allocates from a dedicated Mspace. Mspaces enables good scalability of allocations in multithreaded applications, and it is quite space-efficient. However, the use of the allocated objects by different threads is prone to false sharing, because dlmalloc allocates objects on the same memory page as long as there is free storage on the page. The Page allocator places each object in a separate physical page frame. It implements the primitive approach against false sharing and causes internal fragmentation for object sizes that are not a multiple of page size. The Millipage allocator statically places all objects on Millipages and does not aggregate objects. The Adaptive allocator is based on the Millipage allocator and implements adaptive sharing based on OAGs. To express an allocation scheme's reaction on an access pattern, the number of detected accesses has been measured.

For the first test, the setup consists of two nodes accessing two objects that have been allocated consecutively using varying allocators (see Figure 8.11a). The examined node reads both objects, the second node writes to one object, causing frequent transaction restarts on the examined node. A simple fairness strategy in the storage service ensures that a transaction will commit after restarting once. Figure 8.11a impressively demonstrates that the Mspaces allocator is susceptible to false sharing, whereas the other allocators enable to distinguish both objects. In the second test, a single node accesses two objects conjointly in a loop of 2^{16} transactions. The Page and Millipage allocator detect each access separately. The Mspaces and Adaptive allocator only detect one access per transaction because of spatial locality between the objects.

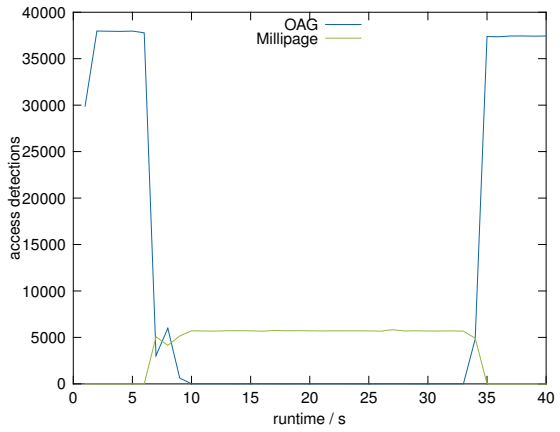
For a synthetic workload with 1024 4-Byte objects, the memory consumption of the Page allocator is severe, whereas the other allocators allocate only the requested object size plus some allocation



(a) number of access detections



(b) memory consumption (logarithmic scale)



(c) dynamic adaptation from OAGs to Millipages and vice versa

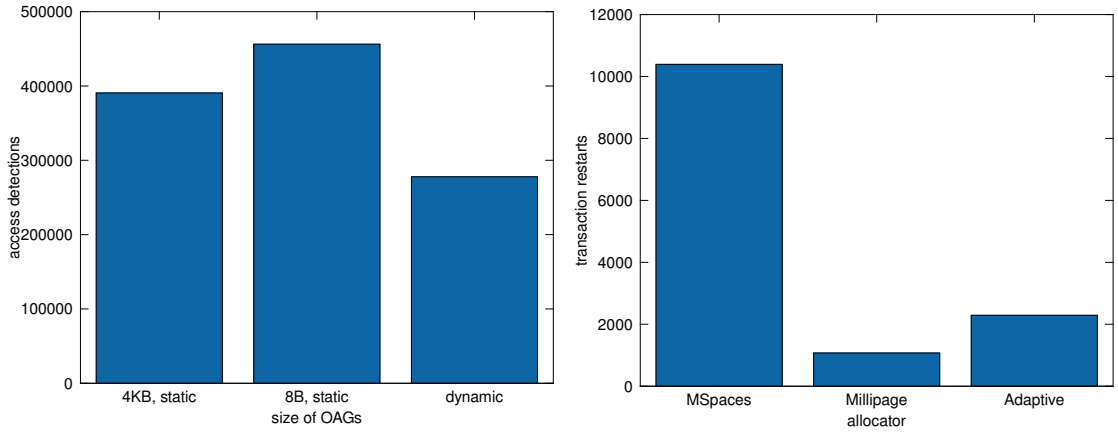
Figure 8.11: Memory consumption and dynamic adaptation of access detections

metadata (see Figure 8.11b). The synthetic workload implies that the standard deviation of the memory consumption equals zero.

Another experiment evaluates how well the granularity adapts to varying object access patterns. The setup consists of two nodes, one of which is running transactions in a loop for 2^{20} times, reading from two objects. About six seconds later, the other node starts up and runs transactions in a loop for 2^{19} times, writing to one of the objects. This access pattern is detected by the object access monitor, which converts the OAG to individual Millipages. After the second node stops modifying the object, the object access monitor aggregates the Millipages to an OAG again. Figure 8.11c subdivides the number of access detections for OAGs and for Millipages. When the second node starts, it does not run transactions at full speed due to frequent conflicts, which causes the first node to switch several times between coarse-granular and fine-granular access detection. OAG-based access detection achieves an access throughput that is a factor of 5 higher compared to Millipage access detection, because larger access units induce less detections. Nonetheless, the splitting of the OAG into Millipages avoids conflicts and enables the nodes to make progress.

False sharing in extended MapReduce

Updates during iterative execution can cause false sharing, as discussed in Chapter 6. In the implementation under examination, continuous word counting operates on a trie where each word is repre-



(a) access detections (b) transaction restarts

Figure 8.12: Word frequency analysis

sented by a path from the tree's root to a node (see Chapter 6). The node at the end of a word stores the frequency of the word it terminates, possibly other statistical information such as time stamps too. Intermediate nodes represent prefixes of a word, storing at most 26 references to next prefix characters. Each node has a size of 216 bytes, which equals 26 references to child nodes plus a 64-Bit counter. When allocating nodes for the trie, the Adaptive allocator splits a physical page frame in 16 Millipages, each 256 bytes large, causing 16% internal fragmentation.

The trie representation of words already counteracts false sharing by enforcing a high fan-out, e.g. compared to a representation of words in a binary tree. Consecutive allocations are satisfied from the same Millipage region, if space allows so. Therefore, graph nodes tend to reside in the same region as their ancestors and descendants, such that grouping adjacent objects makes sense.

In this experiment, each of two worker machines parses a text (the novel *Kim* written by Rudyard Kipling) and counts the individual words. The text consists of 107585 words in total, thereof 10636 different words. Again, the number of access detections is measured, representing how well an allocator makes use of locality (see Figure 8.12a). Additionally, the number of transaction restarts indicates how much access detection suffers from false sharing (see Figure 8.12b). The adaptive access detection mechanism triggers only 60% access detections compared to the Millipage allocator, and it causes less than 25% transaction restarts compared to the Mspaces allocator. Thus, adaptive sharing is not only a theoretical concept, but it can be used efficiently in practice.

8.3 Scalability and storage consumption of in-memory MapReduce

The in-memory MapReduce framework allows to analyze the scalability of MapReduce-structured algorithms in detail.

8.3.1 Performance of map and reduce phases

For the first series of experiments, the problem size is kept constant, such that the amount of work per node was inverse to the number of workers. The applications ran in one-pass MapReduce mode on cluster A.

The raytracer rendered a scene consisting of 228 objects on an image with a dimension of 640x480 pixels. The word-frequency application processed the novel Ulysses by James Joyce, which contains about 268000 words. Two different versions of histogram computation have been implemented and compared. The regular histogram algorithm computes a histogram for exact color values, that is, tu-

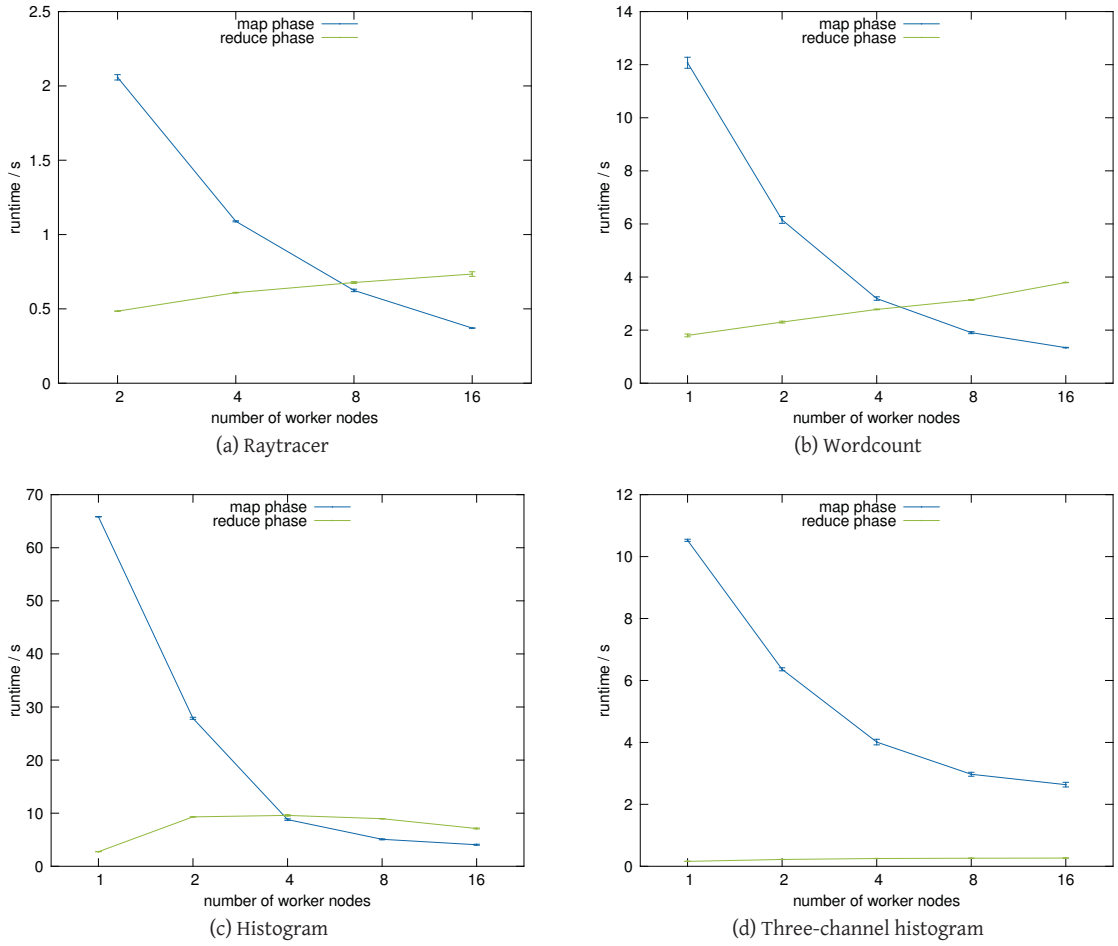
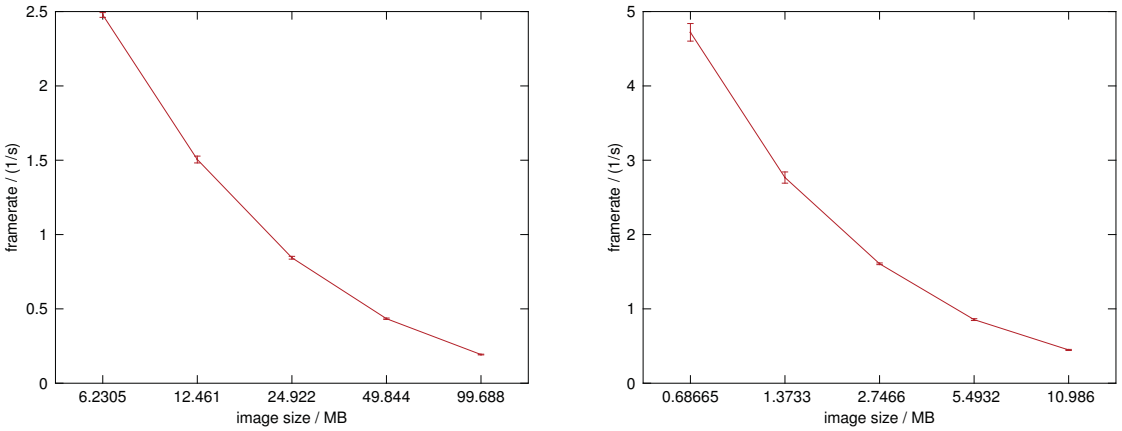


Figure 8.13: Execution times of mapreduce applications

ples of red, green and blue color values. In contrast, the histogram-3 algorithm computes three separate histograms, one for each color channel red, green and blue. Compared to the regular histogram computation, the histogram-3 computation loses more information, but its intermediate and final results are much smaller. The size of a regular histogram in terms of bits per pixel c is $O(2^{3c})$, whereas the size of the three-channel histograms in the histogram-3 application is $O(2^c)$. The regular histogram application ran with an input file of 5.56 MB containing an image of 960x633 pixels and 8 bit per pixel. The histogram-3 application ran on a 24-color bitmap of 6816x5112 pixels, which results in a file size of about 100 MB.

Figures 8.13a, 8.13b, 8.13c and 8.13d show the runtime of raytracer, wordcount and histogram respectively, for different numbers of workers (1 to 32). The storage service used for this measurement does not implement the smart replication discussed in Section 8.2.2. All four applications use the relatively strong transactional consistency model, which limits their scalability for single-pass MapReduce workload, but allows them to be used for extended MapReduce workload. To better understand the overall performance, the runtimes are split for the map and reduce phase. The raytracer reduce phase is rather short compared to its map phase, because it only assembles and stores the image. The wordcount reduce phases do not scale as well as its map phase, a phenomenon caused by the higher conflict rates when merging the intermediate tries to the final trie. The histogram application performs best with 8 worker nodes, because the map phase is still relatively short compared to the transaction overhead.



(a) Histogram (b) Raytracer

Figure 8.14: Framerate of histogram and raytracer application

8.3.2 Performance of iterative operation

To assess the overhead and speedup of iterative MapReduce, the histogram and raytracer application ran with varying problem sizes. The computations were carried out by 16 worker nodes and one master node. Figure 8.14a displays the framerate for histogram computation over an image size that varies from 6.3 MB to 100 MB. Figure 8.14b contains similar data for the raytracer application, with the output image size varying between about 0.7 KB and 10 MB. Higher framerates show also a higher deviation. These measurements confirm the expected inversely proportional relation between image size and framerate.

8.3.3 Performance of framework improvements

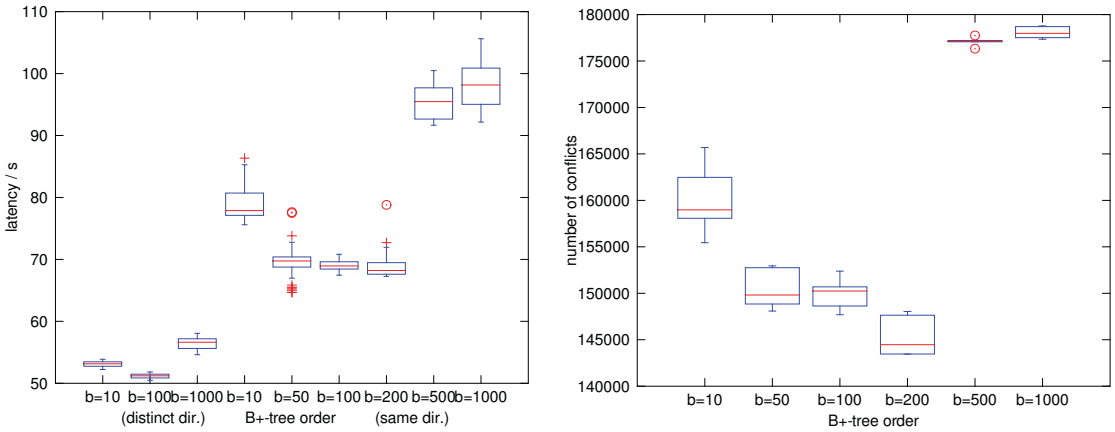
The raytracer application makes a good case for verifying the performance of the work-stealing mechanism described in Section 6.2, because raytracing jobs often deviate in execution time. When executing 320 map jobs on 32 worker nodes, the mean execution time of a single job is 1.4976 seconds with a standard deviation of 0.38466 seconds. Without work-stealing, the average execution time of map jobs per worker node is 14.976 seconds with a standard deviation of 4.3358 seconds. Work-stealing reduced the average execution time of map jobs per worker node to 14.598 seconds and the standard deviation to 0.50582 seconds, a decrease of an order of magnitude.

8.4 In-memory filesystem performance

The performance of the in-memory filesystem has been evaluated using several filesystem microbenchmarks. The experiments were run on 8 nodes from cluster A, each equipped with 2 AMD Opteron 246 processors and 2 GB cache-coherent NUMA RAM. The nodes were configured to boot Debian Squeeze with Linux x86-64 kernel 2.6.32 in disk-less mode via NFS.

8.4.1 Adaptive tree balancing

The first experiment determines the potential of adaptive tree balancing during parallel file creation (see Figure 8.15). Each node created 10,000 files with pseudo-random names, which were generated by computing SHA1 checksums of distinct integer numbers. The files were created either in the same directory by all nodes or in distinct directories per node.



(a) latency

(b) conflict rate (same directory)

Figure 8.15: Parallel file creation

If the filesystem clients write in distinct directories, there are no conflicts, and creating 10,000 files takes between 50 and 60 seconds, depending on the B+-tree order. The tree order $b = 100$ yields better results than $b = 10$ or $b = 1000$. A small tree order causes large transactions because of increased tree height, whereas a large tree order causes a high overhead of data transfers, considering that splitting or coalescing a B+-tree modifies at least half of its entries in the original node and the copy.

However, if all the clients create files in the same directory, the conflict rate is high. The number of conflicts is about half of the total number of committed transactions (approximately 320,000). From tree order 200 to 500, the number of conflicts increases by more than 5%, which is because 200 entries fit into one 4KB access detection unit, but 500 entries require two units.

The results show that a higher tree order does not imply better performance, because the performance benefits of bulk accesses can be overshadowed by increased write set sizes. Therefore, the impact of adaptive tree balancing is limited. Subsection 8.4.3 evaluates the more promising approach of a hash-based distributed filesystem.

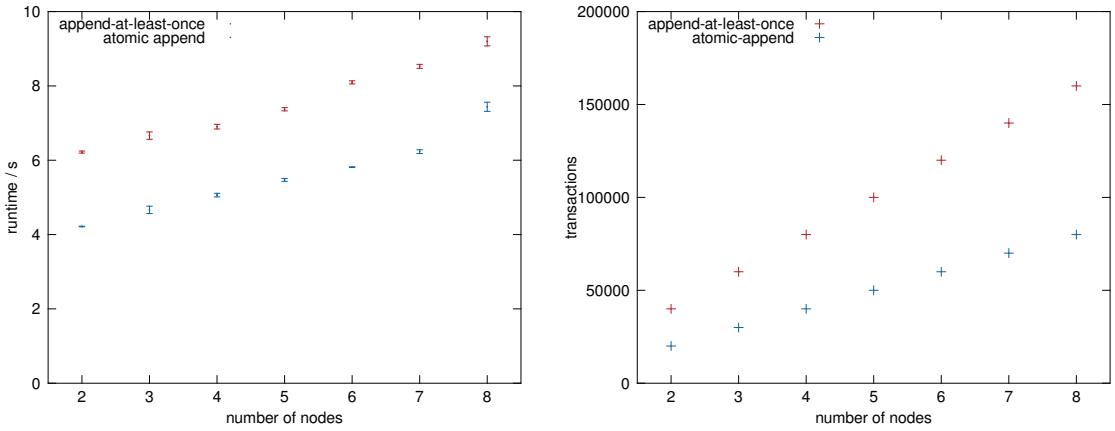
8.4.2 Atomic append

The second experiment demonstrates the benefit of EFS's atomic-append operation. This experiment compares atomic-append with an implementation of append-at-least-once. In both cases, the test application appends 10,000 single 4 KB-blocks to a file. The measurement was done with a distinct node for transaction validation and two to eight nodes mounting and accessing the filesystem. Atomic-append executes 50% faster than the DTM-based append-at-least-once implementation (see Figure 8.16a). The reason for atomic-append's superior performance is that it causes only half the transaction aborts (see Figure 8.16b). Although the userspace implementation of the filesystem requires an additional pair of context switches, the deviation of the results is quite low. The number of transactions executed is fixed, so that no standard deviation is displayed in Figure 8.16b.

The experiments concerning adaptive tree balancing and atomic append show that even a filesystem complying to the operating system's native interface can provide extensions that let slightly modified applications achieve notably better performance.

8.4.3 Keyspace partitioning in hash-based filesystem

Finally the claim about the well-balanced property of the keyspace partitioning of the hash-based filesystem is validated experimentally. A system consisting of a few nodes was setup to run the filesystem with the optimized nameservice discussed in Section 7.4, such that each node inserted 1000 random



(a) performance (b) number of transactions

Figure 8.16: Atomic-append versus append-at-least-once

keys. Afterwards, the number of keys managed by each node was determined. The results have been plotted in Figure 8.17, which states the minimum and maximum number of keys obtained for a single node in function of the total number of nodes, along with the 50th and 90th percentile. The average number of keys stored in a node is unsurprisingly steady at 1000, as few conflicts from the random-number generator are to be expected at this rate, and none from the hash function itself. These results mirror well the expected behavior: In the best case, when the number of nodes is a power of two, every node shares the same amount of keys. Otherwise, some node will assume responsibility for more keys than others, but never more than twice the least managed amount.

In addition, the average time to put and retrieve some data from the hashtable, depending on the number of nodes, has been measured, as well as how this time can vary from a node to another. From the design, one would expect the access times of a single node to be split in two groups: local and remote accesses. A remote access is of course at least as expensive as a local access. Therefore, considering that the more nodes, the more remote accesses, the average access time can be expected to increase slightly with the number of nodes, while being bounded by the access time for a node performing all accesses remotely. The experiments consisted in repeatedly asking every node to perform a few hundreds of the interesting operation. Figure 8.18a plots the average access time for a get access in hashtable in function of the number of nodes, along with the 10th and 90th percentile of the node population in order to check for variations from one node to another. The obtained results justify the expectation of a very moderate increase of the access time when the number of nodes grows. It also shows great homogeneity between the nodes, although the data dispersion increases slightly with more than 22 nodes. The reason for the increased dispersion is that the first 22 nodes are connected over the same network switch, whereas the succeeding nodes communicate over a second switch. The current cost of an access excluding all network communications is moderate, as can be seen from the access time on one node being slightly less than 20 milliseconds. Measuring the time for synchronous put accesses yielded similar results, plotted in Figure 8.18b, although the replication during write operation causes a higher deviation of the results. In conclusion, the design for the hash-based nameservice is sound and allows for efficient implementations, as proven by experimental results.

8.5 Summary

In-memory storage services and applications are influenced by various design decisions. The ECRAM service implements some alternatives and optimizations of in-memory storage design. This chapter has described the evaluation of the contributions presented in the preceding chapters with respect to

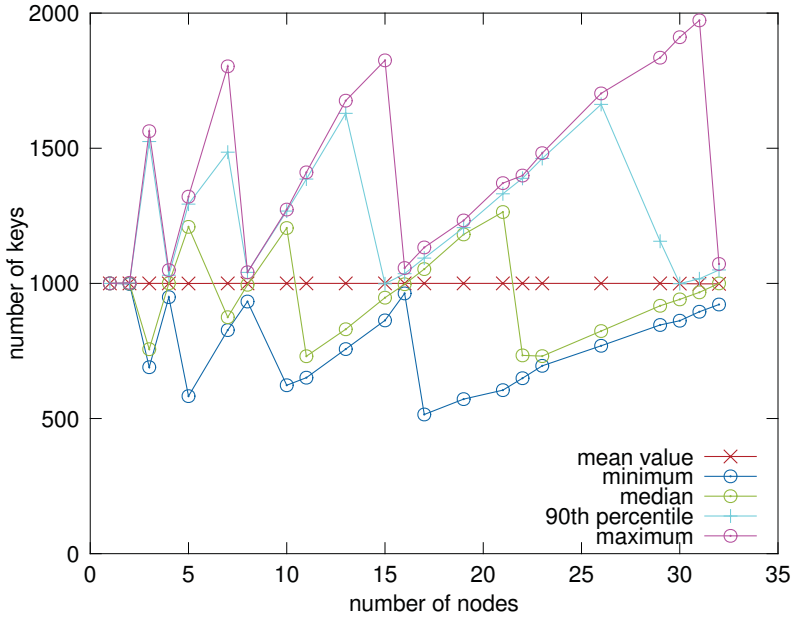
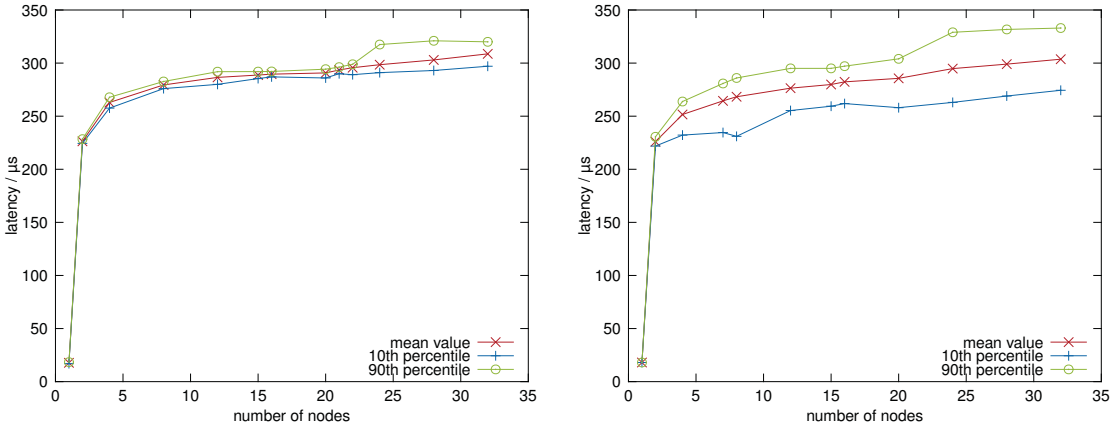


Figure 8.17: Keyspace partitioning in hash-based nameservice



(a) get operation

(b) put operation

Figure 8.18: Timing of operations on hash-based nameservice

runtime performance, scalability in the number of nodes and objects and efficiency of resource usage. The first section has confirmed the observations of other researchers concerning the use of transactional storage versus lock-based synchronization on shared data. The second section has evaluated the adaptive features from Chapters 4 and 5, namely the smart replication protocol and the adaptive sharing granularity. Both features prove useful for all applications under consideration. The third section has focused on the presented in-memory framework for extended MapReduce. The third section has demonstrated that the in-memory storage service can be used efficiently behind a filesystem interface, and that extensions to the native filesystem interface improve the performance and scalability of modified applications. The overall conclusion drawn from the evaluation is that adaptive policies and appropriate interface extensions to tweak replication, consistency and sharing granularity allow applications to achieve better performance in a distributed in-memory storage system.

9

Conclusion

The emerging field of large-scale data analysis pushes parallel and distributed data handling to new limits. Decreasing prices and increasing capacity of volatile memory make it feasible to process large amounts of in-memory data. Instead of persisting data on disk, in-memory storage systems replicate memory over the network to other storage nodes. Developers of distributed storage systems have many choices how to parameterize these systems with respect to data consistency, access latency, availability, and handling of network partitions. This thesis has proposed several adaptive and tunable measures that help storage systems deliver good quality of service without needing much adaptation of applications.

Chapter 2 has described a key-based routing protocol that supports dynamic allocation of storage objects. It has detailed the reliable management of shared metadata and support for different ways to access distributed objects by using memory-mapped objects, stacked allocators and a nameservice built into the storage system.

Chapter 3 has presented a configurable transactional memory consistency model. It has especially emphasized several ways to weaken transactional consistency in favor of increased parallelism.

The observation at the basis of Chapter 4 is that neither a pure invalidation nor a pure update protocol are suited for achieving good scalability of replicating objects. Therefore, the chapter has proposed a smart replication protocol that switches dynamically between invalidating and updating objects. To this end, the protocol monitors object accesses and computes adequate statistics for each storage object.

The subsequent Chapter 5 has discussed the orthogonal issue of deciding an appropriate granularity of object accesses. On the one hand, virtualization of main memory obstructs fine-grained access detection. On the other hand, custom allocators can configure virtual memory in order to adapt the size of object access units during runtime. The chapter has described the implementation of adaptive conflict granularity for distributed transactional memory and additional hints for object size that can be given by the application developer.

In order to substantiate the proposed storage system features, Chapter 6 has introduced the extended MapReduce model that subsumes both iterative and online execution of basic MapReduce. Two practical contributions simplify the implementation of an extended MapReduce framework on distributed transactional storage. First, work queue management can be reduced to synchronization on in-memory condition variables. Second, direct access to in-memory work queues enables load balancing using a work-stealing approach. A range of five applications has illustrated the extended MapReduce model.

In addition to dynamic memory, many conventional applications share data using the operating system's native filesystem interface. Chapter 7 has demonstrated how to implement a distributed filesystem based on in-memory storage. The filesystem architecture integrates well-known concepts such as B+-trees, nameservice and user-level filesystem design. The proposed architecture includes an adaptive mechanism to cache metadata more efficiently. Besides, a special optimization allows slightly modified applications to execute append operations more efficiently. Distributed applications that use a central filesystem often store their files in the same directories. In order to improve the scalability of concurrent accesses to filesystem metadata in the same directory, another approach to represent hierarchical namespaces has been discussed.

In Chapter 8, the contributions made in the preceding chapters were evaluated. The evaluation confirms that the proposed smart and adaptive methods are effective.

In-memory computing is still gaining popularity, spurred by upcoming innovations such as storage class memory and non-volatile main memory. The steadily increasing data volumes necessitate new approaches to tackle storage consistency, access latency and the related issues. Considering the diverging quality-of-service requirements of specific applications, a one-fits-all solution is highly unlikely to exist. A storage service that aims at being used widely must therefore adapt to application requirements, instead of expecting applications to give attention to low-level details of storage organization. This thesis has discussed several contributions for adaptive storage management. Upcoming storage hardware creates yet unknown demands on future storage services. Future research on in-memory computing is most likely to analyze further aspects of adaptive storage management.

Beyond the achievements of this thesis, the topics discussed leave many open fields for future research. This thesis has described some aspects of storage reliability, but the major focus has been the performance of distributed storage systems. Storing all data in volatile memory poses new requirements on redundancy. Major hardware companies are currently introducing non-volatile RAM systems into the market. For the first time since the introduction of spinning disks, these systems will substantially change the conventional storage hierarchy.

The discussed storage service supplements applications with unstructured storage objects. These objects can be used for any purpose, which accommodates convenience and flexibility of application development, but gives the storage service little room for optimization. Structured and typed objects give the service much more information, upon which it can base its decisions concerning sharing granularity and replication.

With the contributions on data sharing granularity and payload data replication, this thesis has discussed mainly the management of payload data. Storage scalability also requires efficient metadata synchronization. Distributed transactional storage has been presented as a means to implement consistent storage operations in a scalable manner. The thesis has not discussed any alternatives to the central validator node. More efficient transaction certification schemes are important especially for high update rates, because then the payload data is no longer the bottleneck for scalability.

Large-scale storage is currently required in diverse application areas, the available hardware is often used inefficiently, and new storage techniques are emerging. These observations indicate that adaptive approaches for distributed in-memory storage have a strong potential to gain importance in the future.

List of Figures

List of Tables

Bibliography

[1] Daniel J. Abadi. Consistency tradeoffs in modern distributed database system design: CAP is only part of the story. *Computer*, 45:37–42, 2012.

[2] Martín Abadi, Tim Harris, and Mojtaba Mehrara. Transactional memory with strong atomicity using off-the-shelf memory protection hardware. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP '09)*, pages 185–196, New York, NY, USA, 2009. ACM.

[3] Martín Abadi, Tim Harris, and Katherine F. Moore. A model of dynamic separation for transactional memory. In *Proceedings of the 19th international conference on Concurrency Theory (CONCUR '08)*, pages 6–20, Berlin, Heidelberg, 2008. Springer-Verlag.

[4] D.A. Abramson and J.L. Keedy. Implementing a large virtual memory in a distributed computing system. In *Proceedings of the 18th Hawaii International Conference on System Sciences*, pages 515–522. IEEE Computer Society, 1985.

[5] Sarita V. Adve and Mark D. Hill. Weak ordering – a new definition. *SIGARCH Comput. Archit. News*, 18(3a):2–14, May 1990.

[6] Atul Adya, Robert Gruber, Barbara Liskov, and Umesh Maheshwari. Efficient optimistic concurrency control using loosely synchronized clocks. In *Proceedings of the 1995 ACM SIGMOD international conference on Management of data (SIGMOD '95)*, pages 23–34, New York, NY, USA, 1995. ACM.

[7] Atul Adya and Barbara Liskov. Lazy consistency using loosely synchronized clocks. In *Proceedings of the sixteenth annual ACM symposium on Principles of distributed computing (PODC '97)*, pages 73–82, New York, NY, USA, 1997. ACM.

[8] Divyakant Agrawal, Amr El Abbadi, Sudipto Das, and Aaron J. Elmore. Database scalability, elasticity, and autonomy in the cloud. In *Proceedings of the 16th international conference on Database systems for advanced applications (DASFAA'11)*, pages 2–15, Berlin, Heidelberg, 2011. Springer-Verlag.

[9] Rakesh Agrawal, Michael J. Carey, and Miron Livny. Concurrency control performance modeling: alternatives and implications. *ACM Trans. Database Syst.*, 12(4):609–654, November 1987.

[10] Marcos K. Aguilera, Arif Merchant, Mehul Shah, Alistair Veitch, and Christos Karamanolis. Sinfonia: A new paradigm for building scalable distributed systems. *ACM Trans. Comput. Syst.*, 27(3):5:1–5:48, November 2009.

[11] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the spring joint computer conference (AFIPS '67)*, pages 483–485, New York, NY, USA, 1967. ACM.

[12] Cristiana Amza, Alan Cox, Karthick Rajamani, and Willy Zwaenepoel. Tradeoffs between false sharing and aggregation in software distributed shared memory. In *Proceedings of the sixth ACM SIGPLAN symposium on Principles and practice of parallel programming (PPOPP '97)*, pages 90–99, New York, NY, USA, 1997. ACM.

[13] Ganesh Ananthanarayanan, Srikanth Kandula, Albert Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Edward Harris. Reining in the outliers in Map-Reduce clusters using Mantri. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation (OSDI'10)*, pages 1–16, Berkeley, CA, USA, 2010. USENIX Association.

[14] Mohammad Ansari, Behram Khan, Mikel Luján, Christos Kotselidis, Chris Kirkham, and Ian Watson. Improving performance by reducing aborts in hardware transactional memory. In *Proceedings of the 5th international conference on High Performance Embedded Architectures and Compilers (HiPEAC'10)*, pages 35–49, Berlin, Heidelberg, 2010. Springer-Verlag.

[15] Mohammad Ansari, Christos Kotselidis, Ian Watson, Chris Kirkham, Mikel Luján, and Kim Jarvis. Lee-tm: A non-trivial benchmark suite for transactional memory. In *Proceedings of the 8th international conference on Algorithms and Architectures for Parallel Processing (ICA3PP '08)*, pages 196–207, Berlin, Heidelberg, 2008. Springer-Verlag.

[16] Mohammad Ansari, Mikel Luján, Christos Kotselidis, Kim Jarvis, Chris Kirkham, and Ian Watson. Robust adaptation to available parallelism in transactional memory applications. *Transactions on High Performance and Embedded Architectures and Compilers*, 3(4), 2008.

[17] G. Antoniu, J.-F. Deverge, and S. Monnet. How to bring together fault tolerance and data consistency to enable grid data sharing: Research articles. *Concurr. Comput. : Pract. Exper.*, 18(13):1705–1723, November 2006.

[18] Ehsan Atoofian, Amirali Baniasadi, and Yvonne Coady. *Adaptive Read Validation in Time-Based Software Transactional Memory*, pages 152–162. Springer-Verlag, Berlin, Heidelberg, 2009.

[19] Henri E. Bal, M. Frans Kaashoek, and Andrew S. Tanenbaum. Orca: A language for parallel programming of distributed systems. *IEEE Trans. Softw. Eng.*, 18(3):190–205, March 1992.

[20] R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1:173–189, 1972. 10.1007/BF00288683.

[21] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Munin: distributed shared memory based on type-specific memory coherence. In *Proceedings of the second ACM SIGPLAN symposium on Principles & practice of parallel programming (PPOPP '90)*, pages 168–176, New York, NY, USA, 1990. ACM.

[22] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: a scalable memory allocator for multithreaded applications. In *Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, pages 117–128, New York, NY, USA, 2000. ACM.

[23] Philip Bernstein and Eric Newcomer. *Principles of transaction processing*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, second edition, 2009.

[24] Philip A. Bernstein and Nathan Goodman. Concurrency control in distributed database systems. *ACM Comput. Surv.*, 13(2):185–221, June 1981.

[25] Philip A. Bernstein, Colin W. Reid, and Sudipto Das. Hyder - a transactional record manager for shared flash. In *Proceedings of the 5th Conference on Innovative Data Systems Research CIDR*, pages 9–20, 2011.

[26] Kemme Bettina and Gustavo Alonso. Database replication: a tale of research across communities. *Proc. VLDB Endow.*, 3:5–12, September 2010.

[27] Annette Bieniusa and Thomas Fuhrmann. Lifting the barriers — reducing latencies with transparent transactional memory. In *Proceedings of the 13th international conference on Distributed Computing and Networking (ICDCN'12)*, pages 16–30, Berlin, Heidelberg, 2012. Springer-Verlag.

[28] K. Birman and T. Joseph. Exploiting virtual synchrony in distributed systems. *SIGOPS Oper. Syst. Rev.*, 21(5):123–138, 1987.

[29] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[30] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: an efficient multithreaded runtime system. *SIGPLAN Not.*, 30:207–216, August 1995.

[31] Robert L. Bocchino, Vikram S. Adve, and Bradford L. Chamberlain. Software transactional memory for large scale clusters. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming (PPoPP '08)*, pages 247–258, New York, NY, USA, 2008. ACM.

[32] Hans-J. Boehm. Reordering constraints for pthread-style locks. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP '07)*, pages 173–182, New York, NY, USA, 2007. ACM.

[33] William J. Bolosky and Michael L. Scott. False sharing and its effect on shared memory performance. In *USENIX Systems on USENIX Experiences with Distributed and Multiprocessor Systems (SEDMS'93)*, pages 3–3, Berkeley, CA, USA, 1993. USENIX Association.

[34] Dhruba Borthakur. *HDFS architecture guide*. http://hadoop.apache.org/common/docs/current/hdfs_design.html. Last accessed June 7, 2012.

[35] Tim Brecht and Harjinder Sandhu. The region trap library: handling traps on application-defined regions of memory. In *Proceedings of the annual conference on USENIX Annual Technical Conference (ATEC '99)*, pages 7–7, Berkeley, CA, USA, 1999. USENIX Association.

[36] Eric A. Brewer. Pushing the CAP: Strategies for consistency and availability. *IEEE Computer*, 45(2):23–29, 2012.

[37] Mihai Burcea, J. Gregory Steffan, and Cristiana Amza. The potential for variable-granularity access tracking for optimistic parallelism. In *Proceedings of the 2008 ACM SIGPLAN workshop on Memory systems performance and correctness (MSPC '08)*, pages 11–15, New York, NY, USA, 2008. ACM.

[38] João Cachopo and António Rito-Silva. Versioned boxes as the basis for memory transactions. *Sci. Comput. Programming*, 63(2):172–185, 2006.

[39] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford transactional applications for multi-processing. In *Proceedings of The IEEE International Symposium on Workload Characterization (IISWC '08)*, September 2008.

[40] Brian David Carlstrom. *Programming with transactional memory*. PhD thesis, Stanford University, Stanford, CA, USA, 2008. AAI3313540.

[41] Fernando Miguel Carvalho and Joao Cachopo. STM with transparent API considered harmful. In *Proceedings of the 11th international conference on Algorithms and architectures for parallel processing (ICA3PP'11)*, pages 326–337, Berlin, Heidelberg, 2011. Springer-Verlag.

[42] Nuno Carvalho, Paolo Romano, and Luís Rodrigues. A generic framework for replicated software transactional memories. In *10th IEEE International Symposium on Network Computing and Applications (NCA 2011)*, pages 271 –274, August 2011.

[43] Nuno Carvalho, Paolo Romano, and Luís Rodrigues. Scert: Speculative certification in replicated software transactional memories. In *Proceedings of the 4th Annual International Conference on Systems and Storage (SYSTOR '11)*, pages 10:1–10:13, New York, NY, USA, 2011. ACM.

[44] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26:4:1–4:26, June 2008.

[45] David Chappell. Introducing Windows Server AppFabric. Technical report, Microsoft Corp., 2010.

[46] Rong Chen, Haibo Chen, and Binyu Zang. Tiled-mapreduce: optimizing resource usages of data-parallel applications on multicore with tiling. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques (PACT '10)*, pages 523–534, New York, NY, USA, 2010. ACM.

[47] David R. Cheriton. Preliminary thoughts on problem-oriented shared memory: a decentralized approach to distributed systems. *SIGOPS Oper. Syst. Rev.*, 19(4):26–33, October 1985.

[48] Weihaw Chuang, Satish Narayanasamy, Ganesh Venkatesh, Jack Sampson, Michael Van Biesbrouck, Gilles Pokam, Brad Calder, and Osvaldo Colavin. Unbounded page-based transactional memory. *SIGPLAN Not.*, 41(11):347–358, 2006.

[49] Douglas Comer. Ubiquitous b-tree. *ACM Comput. Surv.*, 11:121–137, June 1979.

[50] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. MapReduce online. In *Proceedings of the 7th USENIX conference on Networked systems design and implementation (NSDI'10)*, pages 21–21, Berkeley, CA, USA, 2010. USENIX Association.

[51] Intel Corp. *Intel architecture instruction set extensions programming reference*. http://www.intel.com/design/intarch/manuals. Last accessed Nov 28, 2012.

[52] Oracle Corp. *Oracle Coherence documentation*. http://www.oracle.com/technetwork/middleware/coherence/overview/index.html. Last accessed Nov 28, 2012.

[53] Oracle Corp. *Oracle TimesTen documentation*. http://www.oracle.com/technetwork/products/timesten/overview/index.html. Last accessed Nov 28, 2012.

[54] Maria Couceiro, Paolo Romano, Nuno Carvalho, and Luís Rodrigues. D2STM: Dependable distributed software transactional memory. In *Proceedings of the 15th Pacific Rim International Symposium on Dependable Computing (PRDC '09)*, November 2009.

[55] Maria Couceiro, Paolo Romano, and Luis Rodrigues. Polycert: Polymorphic self-optimizing replication for in-memory transactional grids. In *Proceedings of the ACM/IFIP/USENIX 12th Middleware Conference (Middleware'11)*, Lisbon, Portugal, December 2011.

[56] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. G-store: a scalable data store for transactional multi key access in the cloud. In *Proceedings of the 1st ACM symposium on Cloud computing (SoCC '10)*, pages 163–174, New York, NY, USA, 2010. ACM.

[57] Alokika Dash and Brian Demsky. Automatically generating symbolic prefetches for distributed transactional memories. In *Proceedings of the ACM/IFIP/USENIX 11th International Middleware Conference (Middleware'10)*, November 2010.

[58] Alokika Dash and Brian Demsky. Integrating caching and prefetching mechanisms in a distributed transactional memory. *IEEE Transactions on Parallel and Distributed Systems*, 22:1284–1298, 2011.

[59] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation (OSDI'04)*, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[60] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles (SOSP '07)*, pages 205–220, New York, NY, USA, 2007. ACM.

[61] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing (PODC '87)*, pages 1–12, New York, NY, USA, 1987. ACM.

[62] Peter J. Denning. The working set model for program behavior. *Commun. ACM*, 11(5):323–333, 1968.

[63] Peter J. Denning. The locality principle. *Commun. ACM*, 48(7):19–24, 2005.

[64] Peter J. Denning and Stuart C. Schwartz. Properties of the working-set model. *Commun. ACM*, 15(3):191–198, 1972.

[65] BTRFS developers. *BTRFS technical documentation*. https://btrfs.wiki.kernel.org. Last accessed June 7, 2012.

[66] FUSE developers. *FUSE technical documentation*. http://fuse.sourceforge.net/. Last accessed June 7, 2012.

[67] David J DeWitt, Randy H Katz, Frank Olken, Leonard D Shapiro, Michael R Stonebraker, and David A. Wood. Implementation techniques for main memory database systems. *SIGMOD Rec.*, 14(2):1–8, June 1984.

[68] Dave Dice and Alex Garthwaite. Mostly lock-free malloc. In *ISMM '02: Proceedings of the 3rd international symposium on Memory management*, pages 163–174, New York, NY, USA, 2002. ACM.

[69] Dave Dice, Yossi Lev, Virendra J. Marathe, Mark Moir, Dan Nussbaum, and Marek Olszewski. Simplifying concurrent algorithms by exploiting hardware transactional memory. In *Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures (SPAA '10)*, pages 325–334, New York, NY, USA, 2010. ACM.

[70] Dave Dice, Yossi Lev, Mark Moir, and Daniel Nussbaum. Early experience with a commercial hardware transactional memory implementation. *SIGPLAN Not.*, 44(3):157–168, March 2009.

[71] Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking II. In *Proceedings of the 20th international conference on Distributed Computing*, DISC'06, pages 194–208, Berlin, Heidelberg, 2006. Springer-Verlag.

[72] M. Dubois, C. Scheurich, and F. Briggs. Memory access buffering in multiprocessors. *SIGARCH Comput. Archit. News*, 14:434–442, May 1986.

[73] Paul R. Eggert and Douglas Stott Parker Jr. File systems in user space. In *Proceedings of the USENIX Winter 1993 Technical Conference*, pages 229–240, 1993.

[74] Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, and Geoffrey Fox. Twister: a runtime for iterative mapreduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 810–818, New York, NY, USA, 2010. ACM.

[75] Carla Schlatter Ellis and Thomas J. Olson. Algorithms for parallel memory allocation. *Int. J. Parallel Program.*, 17:303–345, August 1989.

[76] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. SAP HANA database: data management for modern business applications. *SIGMOD Rec.*, 40(4):45–51, January 2012.

[77] Alan D. Fekete and Krithi Ramamritham. Consistency models for replicated data. In *Replication*, pages 1–17. Springer-Verlag, Berlin, Heidelberg, 2010.

[78] Pascal Felber, Christof Fetzer, and Torvald Riegel. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming (PPoPP '08)*, pages 237–246, New York, NY, USA, 2008. ACM.

[79] Sérgio Miguel Fernandes and João Cachopo. Lock-free and scalable multi-version software transactional memory. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 179–188, New York, NY, USA, 2011. ACM.

[80] Christof Fetzer and Martin Süßkraut. Switchblade: enforcing dynamic personalized system call models. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008 (Eurosys '08)*, pages 273–286, New York, NY, USA, 2008. ACM.

[81] B. Fleisch and G. Popek. Mirage: a coherent distributed shared memory design. *SIGOPS Oper. Syst. Rev.*, 23(5):211–223, November 1989.

[82] A. Forestiero, E. Leonardi, C. Mastroianni, and M. Meo. Self-Chord: A bio-inspired P2P framework for self-organizing distributed systems. *IEEE/ACM Transactions on Networking*, 18(5):1651 –1664, October 2010.

[83] Armando Fox and Eric A. Brewer. Harvest, yield, and scalable tolerant systems. In *Proceedings of the The Seventh Workshop on Hot Topics in Operating Systems (HOTOS '99)*, page 174, Washington, DC, USA, 1999. IEEE Computer Society.

[84] Armando Fox, Steven D. Gribble, Yatin Chawathe, Eric A. Brewer, and Paul Gauthier. Cluster-based scalable network services. *SIGOPS Oper. Syst. Rev.*, 31(5):78–91, 1997.

[85] Geoffrey C. Fox, Roy D. Williams, and Paul C. Messina. *Parallel Programming Works*. Morgan Kaufmann Publishers, San Fransisco, CA, USA, 1994.

[86] H. Garcia-Molina. Elections in a distributed computing system. *IEEE Trans. Comput.*, 31(1):48–59, January 1982.

[87] H. Garcia-Molina and K. Salem. Main memory database systems: An overview. *IEEE Trans. on Knowl. and Data Eng.*, 4(6):509–516, December 1992.

[88] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. *SIGOPS Oper. Syst. Rev.*, 37:29–43, October 2003.

[89] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, 2002.

[90] Ralph Harry Goeckelmann. *Speicherverwaltung und Bootstrategien für ein Betriebssystem mit transaktionalem verteilten Heap*. PhD thesis, Universität Ulm, 2005.

[91] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A secure environment for untrusted helper applications confining the wily hacker. In *Proceedings of the 6th conference on USENIX Security Symposium, Focusing on Applications of Cryptography - Volume 6*, pages 1–1, Berkeley, CA, USA, 1996. USENIX Association.

[92] Jim Gray. The transaction concept: virtues and limitations (invited paper). In *Proceedings of the seventh international conference on Very Large Data Bases (VLDB '1981)*, pages 144–154. VLDB Endowment, 1981.

[93] Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. The dangers of replication and a solution. In *Proceedings of the 1996 ACM SIGMOD international conference on Management of data (SIGMOD '96)*, pages 173–182, New York, NY, USA, 1996. ACM.

[94] Rachid Guerraoui and Michal Kapalka. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming (PPoPP '08)*, pages 175–184, New York, NY, USA, 2008. ACM.

[95] L. Hammond, V. Wong, M. Chen, B.D. Carlstrom, J.D. Davis, B. Hertzberg, M.K. Prabhu, Honggo Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, pages 102 – 113, June 2004.

[96] Derin Harmanci, Pascal Felber, Vincent Gramoli, and Christof Fetzer. TMunit: Testing transactional memories. In *4th Workshop on Transactional Computing (TRANSACT '09)*, February 2009.

[97] Tim Harris, James R. Larus, and Ravi Rajwar. *Transactional Memory, 2nd edition*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2010.

[98] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011.

[99] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, 21(2):289–300, 1993.

[100] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[101] C. A. R. Hoare. Monitors: an operating system structuring concept. *Commun. ACM*, 17(10):549–557, 1974.

[102] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and performance in a distributed file system. *ACM Trans. Comput. Syst.*, 6(1):51–81, February 1988.

[103] Z. Huang, C. Sun, M. Purvis, and S. Cranefield. View-based consistency and false sharing effect in distributed shared memory. *SIGOPS Oper. Syst. Rev.*, 35(2):51–60, 2001.

[104] Felix Hupfeld, Toni Cortes, Björn Kolbeck, Jan Stender, Erich Focht, Matthias Hess, Jesus Malo, Jonathan Marti, and Eugenio Cesario. The XtreemFS architecture – a case for object-based file systems in grids. *Concurr. Comput. : Pract. Exper.*, 20(17):2049–2060, December 2008.

[105] Liviu Iftode, Jaswinder Pal Singh, and Kai Li. Scope consistency: a bridge between release consistency and entry consistency. In *Proceedings of the eighth annual ACM symposium on Parallel algorithms and architectures (SPAA '96)*, pages 277–287, New York, NY, USA, 1996. ACM.

[106] GigaSpaces Technologies Inc. *GigaSpaces XAP documentation*. http://wiki.gigaspaces.com. Last accessed Nov 28, 2012.

[107] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *SIGOPS Oper. Syst. Rev.*, 41:59–72, March 2007.

[108] Ayal Itzkovitz and Assaf Schuster. Multiview and millipage – fine-grain sharing in page-based dsms. In *Proceedings of the third symposium on Operating systems design and implementation (OSDI '99)*, pages 215–228, Berkeley, CA, USA, 1999. USENIX Association.

[109] Ramakrishna Karedla, J. Spencer Love, and Bradley G. Wherry. Caching strategies to improve disk system performance. *Computer*, 27:38–46, 1994.

[110] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing (STOC '97)*, pages 654–663, New York, NY, USA, 1997. ACM.

[111] David R. Karger and Matthias Ruhl. Simple efficient load balancing algorithms for peer-to-peer systems. In *Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures (SPAA '04)*, pages 36–43, New York, NY, USA, 2004. ACM.

[112] Pierre Karpman. Metadata management for EIS. Technical report, Universität Düsseldorf, 2011. September 5, 2011.

[113] Pete Keleher, Alan L. Cox, and Willy Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proc. of the 19th Annual Int'l Symp. on Computer Architecture (ISCA'92)*, pages 13–21, 1992.

[114] Behram Khan, Matthew Horsnell, Ian Rogers, Mikel Luján, Andrew Dinn, and Ian Watson. An object-aware hardware transactional memory. In *Proceedings of International Conference on High Performance Computing and Communications (HPCC)*, pages 51–58, 2008.

[115] Predrag Knezevic, Andreas Wombacher, and Thomas Risse. Highly available DHTs: Keeping data consistency after updates. In *Proceedings of the Fourth Conference on Agents and Peer-to-Peer Computing (AP2PC)*, pages 70–80, 2005.

[116] Kenneth C. Knowlton. A fast storage allocator. *Commun. ACM*, 8:623–624, October 1965.

[117] Donald E. Knuth. *The art of computer programming, volume 3 (2nd ed.): sorting and searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

[118] Guy Korland, Nir Shavit, and Pascal Felber. Deuce: Noninvasive software transactional memory in Java. *Transactions on HiPEAC*, 5(2), 2010.

[119] Christos Kotselidis, Mohammad Ansari, Kim Jarvis, Mikel Luján, Chris Kirkham, and Ian Watson. Distm: A software transactional memory framework for clusters. In *Proceedings of the 37th IEEE International Conference on Parallel Processing (ICPP '08)*. IEEE Computer Society Press, September 2008.

[120] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Chris Wells, and Ben Zhao. OceanStore: an architecture for global-scale persistent storage. *SIGARCH Comput. Archit. News*, 28(5):190–201, November 2000.

[121] H. T. Kung and John T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, 1981.

[122] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44:35–40, April 2010.

[123] Ralf Lämmel. Google's MapReduce programming model – revisited. *Sci. Comput. Program.*, 70(1):1–30, 2008.

[124] D. Lea. A memory allocator. Technical report, State University of New York at Oswego, 2000. http://gee.cs.oswego.edu/dl/html/malloc.html. Last accessed Nov 28, 2012.

[125] Victor C. S. Lee and Kwok-Wa Lam. Optimistic concurrency control in broadcast environments: Looking forward at the server and backward at the clients. In *Proceedings of the First International Conference on Mobile Data Access (MDA '99)*, pages 97–106, London, UK, 1999. Springer-Verlag.

[126] Philip L. Lehman and s. Bing Yao. Efficient locking for concurrent operations on b-trees. *ACM Trans. Database Syst.*, 6:650–670, December 1981.

[127] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. Comput. Syst.*, 7:321–359, November 1989.

[128] Jochen Liedtke. L4 Nucleus Version X reference manual, 1999. http://www.l4hq.org/. Last accessed Nov 28, 2012.

[129] Barbara Liskov, Mark Day, and Liuba Shrira. Distributed object management in Thor. In *Proceedings of the International Workshop on Distributed Object Management IWDOM*, pages 79–91, 1992.

[130] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP '11)*, pages 401–416, New York, NY, USA, 2011. ACM.

[131] Umesh Maheshwari and Barbara Liskov. Collecting distributed garbage cycles by back tracing. In *Proceedings of the sixteenth annual ACM symposium on Principles of distributed computing (PODC '97)*, pages 239–248, New York, NY, USA, 1997. ACM.

[132] Kaloian Manassiev, Madalin Mihailescu, and Cristiana Amza. Exploiting distributed version concurrency in a transactional memory cluster. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP '06)*, pages 198–208, New York, NY, USA, 2006. ACM.

[133] Milo Martin, Colin Blundell, and E. Lewis. Subtleties of transactional memory atomicity semantics. *IEEE Comput. Archit. Lett.*, 5(2):17, 2006.

[134] Michael Matz, Jan Hubiçka, Andreas Jaeger, and Mark Mitchell. *AMD64 Architecture Processor Supplement to the System V Application Binary Interface*, draft version 0.99.5 edition, September 2010. http://www.x86-64.org. Last accessed Nov 28, 2012.

[135] Paul E. McKenney, Maged M. Michael, Josh Triplett, and Jonathan Walpole. Why the grass may not be greener on the other side: a comparison of locking vs. transactional memory. *SIGOPS Oper. Syst. Rev.*, 44(3):93–101, 2010.

[136] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman. *The design and implementation of the 4.4BSD operating system*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1996.

[137] Scott McLean, Kim Williams, and James Naftel. *Microsoft .Net Remoting*. Microsoft Press, Redmond, WA, USA, 2002.

[138] C. Metz. IP anycast point-to-(any) point communication. *Internet Computing, IEEE*, 6(2):94 –98, March 2002.

[139] Michelle J. Moravan, Jayaram Bobba, Kevin E. Moore, Luke Yen, Mark D. Hill, Ben Liblit, Michael M. Swift, and David A. Wood. Supporting nested transactional memory in logTM. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems (ASPLOS-XII)*, pages 359–370, New York, NY, USA, 2006. ACM.

[140] David Mosberger. Memory consistency models. *SIGOPS Oper. Syst. Rev.*, 27:18–26, January 1993.

[141] Kim-Thomas Möller, Marc-Florian Müller, Michael Sonnenfroh, and Michael Schöttner. A software transactional memory service for grids. In *International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP 2009)*, 2009.

[142] Marc-Florian Müller. *Transaktionale replizierte Objekte für verteilte und parallele Anwendungen*. PhD thesis, Mathematisch-Naturwissenschaftliche Fakultät der Heinrich-Heine-Universität Düsseldorf, Düsseldorf, 2011.

[143] Yang Ni, Vijay S. Menon, Ali-Reza Adl-Tabatabai, Antony L. Hosking, Richard L. Hudson, J. Eliot B. Moss, Bratin Saha, and Tatiana Shpeisman. Open nesting in software transactional memory. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP '07)*, pages 68–78, New York, NY, USA, 2007. ACM.

[144] Bogdan Nicolae, Gabriel Antoniu, Luc Bougé, Diana Moise, and Alexandra Carpen-Amarie. BlobSeer: Next-generation data management for large scale infrastructures. *J. Parallel Distrib. Comput.*, 71:169–184, February 2011.

[145] Nitzan Niv and Assaf Schuster. Transparent adaptation of sharing granularity in MultiView-based DSM systems. *Softw. Pract. Exper.*, 31(15):1439–1459, 2001.

[146] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Diego Ongaro, Guru Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. The case for RAMCloud. *Commun. ACM*, 54(7):121–130, July 2011.

[147] Victor Pankratius and Ali-Reza Adl-Tabatabai. A study of transactional memory vs. locks in practice. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures (SPAA '11)*, pages 43–52, New York, NY, USA, 2011. ACM.

[148] Daniel Peng and Frank Dabek. Large-scale incremental processing using distributed transactions and notifications. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation (OSDI'10)*, pages 1–15, Berkeley, CA, USA, 2010. USENIX Association.

[149] Dmitri Perelman, Anton Byshevsky, Oleg Litmanovich, and Idit Keidar. SMV: Selective multiversioning STM. In David Peleg, editor, *Distributed Computing*, volume 6950 of *Lecture Notes in Computer Science*, pages 125–140. Springer Berlin / Heidelberg, 2011. 10.1007/978-3-642-24100-09.

[150] C. Greg Plaxton, Rajmohan Rajaraman, and Andréa W. Richa. Accessing nearby copies of replicated objects in a distributed environment. *Theory Comput. Syst.*, pages 241–280, 1999.

[151] Dan R. K. Ports, Austin T. Clements, Irene Zhang, Samuel Madden, and Barbara Liskov. Transactional consistency and automatic management in an application data cache. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI '10)*, Vancouver, BC, Canada, October 2010. USENIX.

[152] Harald Prokop. *Cache-Oblivious Algorithms*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 1999.

[153] Sean Quinlan and Sean Dorward. Venti: A new approach to archival storage. In *Proceedings of the Conference on File and Storage Technologies (FAST '02)*, pages 89–101, Berkeley, CA, USA, 2002. USENIX Association.

[154] Raghu Ramakrishnan. CAP and cloud data management. *Computer*, 45:43–49, 2012.

[155] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. Evaluating MapReduce for multi-core and multiprocessor systems. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 13–24, Washington, DC, USA, 2007. IEEE Computer Society.

[156] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. *SIGCOMM Comput. Commun. Rev.*, 31:161–172, August 2001.

[157] D. P. Reed. *Naming and synchronization in a decentralized computer system*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 1978.

[158] Kim-Thomas Rehmann, Kevin Beineke, and Michael Schöttner. Smart replication for in-memory computations. In *Proceedings of the Eighteenth IEEE International Conference on Parallel and Distributed Systems 2012 (ICPADS 2012)*, Singapore, December 2012.

[159] Kim-Thomas Rehmann, Serdar Dere, and Michael Schöttner. Adaptive meta-data management and flexible consistency in a distributed in-memory file-system. In *Proceedings of the Twelfth International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT 2011)*, Gwangju, Korea, October 2011.

[160] Kim-Thomas Rehmann, Marc-Florian Müller, and Michael Schöttner. Adaptive conflict unit size for distributed optimistic synchronization. In *Proceedings of The Sixteenth International Conference on Parallel Computing (Euro-Par 2010)*, Ischia, Naples, Italy, August 2010.

[161] Kim-Thomas Rehmann and Michael Schöttner. Applications and evaluation of in-memory MapReduce. In *Proceedings of the Third International IEEE Conference on Cloud Computing Technology and Science 2011 (CloudCom 2011)*, Athens, Greece, December 2011.

[162] Kim-Thomas Rehmann and Michael Schöttner. An in-memory framework for extended mapreduce. In *Proceedings of the Seventeenth IEEE International Conference on Parallel and Distributed Systems 2011 (ICPADS 2011)*, Tainan, Taiwan, December 2011.

[163] R. F. Resende and A. El Abbadi. On the serializability theorem for nested transactions. *Information Processing Letters*, 50(4):177 – 183, 1994.

[164] Sean Rhea, Dennis Geels, Timothy Roscoe, and John Kubiatowicz. Handling churn in a DHT. In *Proceedings of the annual conference on USENIX Annual Technical Conference (ATEC '04)*, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[165] Torvald Riegel, Pascal Felber, and Christof Fetzer. A lazy snapshot algorithm with eager validation. In *In Proceedings of the 20th International Symposium on Distributed Computing (DISC'06)*, pages 284–298, 2006.

[166] Torvald Riegel, Christof Fetzer, and Pascal Felber. Time-based transactional memory with scalable time bases. In *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures (SPAA '07)*, pages 221–228, New York, NY, USA, 2007. ACM.

[167] Ohad Rodeh and Avi Teperman. zFS: A scalable distributed file system using object disks. In *Proceedings of the 20 th IEEE/11 th NASA Goddard Conference on Mass Storage Systems and Technologies (MSS'03)*, pages 207–, Washington, DC, USA, 2003. IEEE Computer Society.

[168] Michael D. Rogers, Christopher Diaz, Raphael Finkel, James Griffioen, and James E. Lumpp. BTMD: Small, fast diffs for WAN-based DSM. In *Proceedings of the Second International Workshop on Software Distributed Shared Memory*, 2000.

[169] Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg (Middleware '01)*, pages 329–350, London, UK, 2001. Springer-Verlag.

[170] Pedro Ruivo, Maria Couceiro, Paolo Romano, and Luis Rodrigues. Exploiting total order multicast in weakly consistent transactional caches. In *Proceedings of the 17th Pacific Rim International Symposium on Dependable Computing (PRDC'11)*, Pasadena, California, USA, December 2011.

[171] Yasushi Saito and Marc Shapiro. Optimistic replication. *ACM Comput. Surv.*, 37(1):42–81, 2005.

[172] R. Sandberg, D. Golgberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the SUN network filesystem. In *Innovations in Internetworking*, pages 379–390. Artech House, Inc., Norwood, MA, USA, 1988.

[173] M. Satyanarayanan. The evolution of Coda. *ACM Trans. Comput. Syst.*, 20(2):85–124, May 2002.

[174] Scott Schneider, Christos D. Antonopoulos, and Dimitrios S. Nikolopoulos. Scalable locality-conscious multithreaded memory allocation. In *Proceedings of the 5th international symposium on Memory management (ISMM '06)*, pages 84–94, New York, NY, USA, 2006. ACM.

[175] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing (PODC '95)*, pages 204–213, New York, NY, USA, 1995. ACM.

[176] Liuba Shrira, Barbara Liskov, Miguel Castro, and Atul Adya. How to scale transactional storage systems. In *Proceedings of the 7th workshop on ACM SIGOPS European workshop*, pages 121–127, New York, NY, USA, 1996. ACM.

[177] Michael Sonnenfroh. *Ein datenzentriertes Programmiermodell für verteilte virtuelle Welten*. PhD thesis, Mathematisch-Naturwissenschaftliche Fakultät der Heinrich-Heine-Universität Düsseldorf, 2010.

[178] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *SIGCOMM Comput. Commun. Rev.*, 31:149–160, August 2001.

[179] Michael Stonebraker. The case for shared nothing. *IEEE Database Eng. Bull.*, 9(1):4–9, 1986.

[180] Justin Talbot, Richard M. Yoo, and Christos Kozyrakis. Phoenix++: modular MapReduce for shared-memory systems. In *Proceedings of the second international workshop on MapReduce and its applications (MapReduce '11)*, pages 9–16, New York, NY, USA, 2011. ACM.

[181] A.S. Tanenbaum and M. Steen. *Distributed systems: principles and paradigms*. Pearson Prentice Hall, 2007.

[182] Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike Spreitzer, Marvin Theimer, and Brent W. Welch. Session guarantees for weakly consistent replicated data. In *Proceedings of the Third International Conference on Parallel and Distributed Information Systems (PDIS '94)*, pages 140–149, Washington, DC, USA, 1994. IEEE Computer Society.

[183] The IEEE and The Open Group. *The Open Group Base Specifications Issue 6 – IEEE Std 1003.1, 2004 Edition*. IEEE, New York, NY, USA, 2004.

[184] Josep Torrellas, Monica S. Lam, and John L. Hennessy. False sharing and spatial locality in multiprocessor caches. *IEEE Trans. Computers*, 43(6):651–663, 1994.

[185] Robbert van Renesse and Rachid Guerraoui. *Replication Techniques for Availability*, pages 10–40. Springer-Verlag, Berlin, Heidelberg, 2010.

[186] Maarten van Steen and Guillaume Pierre. *Replicating for performance: case studies*, pages 73–89. Springer-Verlag, Berlin, Heidelberg, 2010.

[187] Steven P. Vanderwiel and David J. Lilja. Data prefetch mechanisms. *ACM Comput. Surv.*, 32(2):174–199, June 2000.

[188] Steve Vinoski. New features for CORBA 3.0. *Commun. ACM*, 41(10):44–52, 1998.

[189] Werner Vogels. Eventually consistent. *Commun. ACM*, 52(1):40–44, 2009.

[190] Haris Volos, Andres Jaan Tack, Neelam Goyal, Michael M. Swift, and Adam Welc. xCalls: safe I/O in memory transactions. In *Proceedings of the 4th ACM European conference on Computer systems (EuroSys '09)*, pages 247–260, New York, NY, USA, 2009. ACM.

[191] Paul Wang and William E. Weihl. Scalable concurrent B-trees using multi-version memory. *Journal of Parallel and Distributed Computing*, 32(1):28 – 48, 1996.

[192] Gerhard Weikum and Hans-J. Schek. *Concepts and applications of multilevel transactions and open nested transactions*, pages 515–553. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992.

[193] Gerhard Weikum and Gottfried Vossen. *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[194] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In *Proceedings of the International Workshop on Memory Management (IWMM '95)*, pages 1–116, London, UK, 1995. Springer-Verlag.

[195] Richard M. Yoo, Anthony Romano, and Christos Kozyrakis. Phoenix rebirth: Scalable MapReduce on a large-scale shared-memory system. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC '09)*, pages 198–207, Washington, DC, USA, 2009. IEEE Computer Society.

[196] Bingjing Zhang, Yang Ruan, Tak-Lon Wu, J. Qiu, A. Hughes, and G. Fox. Applying Twister to scientific applications. In *2010 IEEE Second International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 25 –32, December 2010.

[197] Ben Y. Zhao, John D. Kubiatowicz, and Anthony D. Joseph. Tapestry: a fault-tolerant wide-area application infrastructure. *SIGCOMM Comput. Commun. Rev.*, 32:81–81, January 2002.

[198] Weizhong Zhao, Huifang Ma, and Qing He. Parallel k-means clustering based on MapReduce. In *Proceedings of the 1st International Conference on Cloud Computing (CloudCom '09)*, pages 674–679, Berlin, Heidelberg, 2009. Springer-Verlag.

[199] Yuanyuan Zhou, Zhifeng Chen, and Kai Li. Second-level buffer cache management. *IEEE Transactions on Parallel and Distributed Systems*, 15:505–519, 2004.

Publication Record of the Author

[1] Kim-Thomas Rehmann, Kevin Beineke, and Michael Schöttner. Smart replication for in-memory computations. In *Proceedings of the Eighteenth IEEE International Conference on Parallel and Distributed Systems 2012 (ICPADS 2012)*, Singapore, December 2012. (acceptance rate 29.6%)

[2] Kim-Thomas Rehmann and Michael Schöttner. Applications and evaluation of in-memory MapReduce. In *Proceedings of the Third International IEEE Conference on Cloud Computing Technology and Science 2011 (CloudCom 2011)*, Athens, Greece, December 2011. (acceptance rate 24%)

[3] Kim-Thomas Rehmann and Michael Schöttner. An in-memory framework for extended mapreduce. In *Proceedings of the Seventeenth IEEE International Conference on Parallel and Distributed Systems 2011 (ICPADS 2011)*, Tainan, Taiwan, December 2011. (acceptance rate 27%)

[4] Kim-Thomas Rehmann, Serdar Dere, and Michael Schöttner. Adaptive meta-data management and flexible consistency in a distributed in-memory file-system. In *Proceedings of the Twelfth International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT 2011)*, Gwangju, Korea, October 2011. (acceptance rate 27.9%)

[5] Marc-Florian Müller, Kim-Thomas Möller, and Michael Schöttner. Commit protocols for a distributed transactional memory. In *Proceedings of the Eleventh International Conference on Parallel and Distributed Computing, Applications, and Technologies (PDCAT 2010)*, Wuhan, China, December 2010.

[6] Kim-Thomas Rehmann, Marc-Florian Müller, and Michael Schöttner. Adaptive conflict unit size for distributed optimistic synchronization. In *Proceedings of The Sixteenth International Conference on Parallel Computing (Euro-Par 2010)*, Ischia, Naples, Italy, August 2010. (acceptance rate 35.1%)

[7] Marc-Florian Müller, Kim-Thomas Möller, and Michael Schöttner. Efficient commit ordering of speculative transactions. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications 2010 (PDPTA 2010)*, Las Vegas, NV, USA, 2010.

[8] Michael Sonnenfroh, Marc-Florian Müller, Kim-Thomas Möller, and Michael Schöttner. Speculative transactions for distributed interactive applications. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications 2010 (PDPTA 2010)*, Las Vegas, NV, USA, 2010.

[9] Kim-Thomas Möller, Marc-Florian Müller, Michael Sonnenfroh, and Michael Schöttner. A software transactional memory service for grids. In *ICA3PP 2009: International Conference on Algorithms and Architectures for Parallel Processing*, Taipei, Taiwan, 2009.

[10] Marc-Florian Müller, Kim-Thomas Möller, Michael Sonnenfroh, and Michael Schöttner. Transactional data sharing in grids. In *Proceedings of the International Conference on Parallel and Distributed Computing and Systems 2008 (PDCS 2008)*, Orlando, FL, USA, 2008.

Index

A

ECRAM Application Programming Interface

A.1 Introduction

To help finding a way through the source code, this document specifies the names of files and functions. Preprocessor definitions that can be set in the configuration dialog (`make menuconfig`) are specified in footnotes.

First, we present the ECRAM library's interface. Second, we give a general introduction to developing applications with ECRAM. Third, we present the internals of ECRAM's various components and modules. Early versions of this reference were co-authored with contributors of the ECRAM project.

A.2 ECRAM Interface

The ECRAM interface is defined in file `ecram.h`. A node participates in a distributed application by using the functions `ecram_startup` and `ecram_shutdown`. When a bootstrap node is not specified, the node starts a distributed application as the first node. Otherwise, it tries to join an already running distributed application.

A.2.1 Objects

An ECRAM object is identified by an object ID (OID of type `ecram_object_id_t`) that is unique in the scope of a distributed application. The width of OIDs is configurable at compile-time.[1]

ECRAM either supports direct-mapped objects or flexible objects.[2] Direct-mapped objects reside in the CPU's virtual address space, such that their OIDs coincide with their virtual address. Flexible objects are not permanently associated with virtual memory addresses.

The characeristics of direct-mapped objects are as follows:

- OIDs are 64 bit wide.

[1] config parameter `ECRAM_OBJECT_ID`
[2] config parameter `ECRAM_IN_MEMORY_OBJECTS`

- Atomic objects are 1 byte large. OIDs of atomic objects are consecutive, i.e. offset 1 from OID x is object x+1.
- An OID is a memory address.
- Accesses are transparently detected via MMU.[3] Alternatively, the application can use `ecram_read/ecram_write` for explicit accesses.[4]

The characeristics of flexible objects are as follows:

- OID width is not restricted.
- Objects have variable size. OIDs of atomic objects are not necessarily consecutive. This is not fully implemented yet.
- An OID is independent of object storage.
- The functions `ecram_read/ecram_write` must be used to access objects.[5] Access detection via MMU is not supported.

Objects are created using `ecram_alloc` and destroyed using `ecram_free`. Files can be mapped as objects with `ecram_mmap` similarly to the `mmap` system call.[6] The `ecram_munmap` function deletes an object that has been mapped using `ecram_mmap`, but currently it does not synchronize the object with the file. The function `ecram_msync` is not implemented yet, because file mappings are not managed globally.

A.2.2 Consistency

Applications can start transactions with `ecram_bot` and finish them with `ecram_eot`. ECRAM executes transactions speculatively. If ECRAM detects a conflict with a concurrent transaction, it transparently restarts the transaction.[7] The semantics of non-transparent restart are still undefined. The restart mechanism can optionally restart the CPU's floating-point unit.[8] An extended library interface could allow weakly consistent object accesses.[9]

To allow experimenting with transaction properties, the `ecram_transaction_attributes_t` structure enables setting various attributes in calls to `ecram_bot`. For example, transaction statistics can be exported to the calling application.[10] Access to transaction statistics is also possible using the function `ecram_get_statistics`.[11] The validation phase can optionally be skipped if the developer can preclude or tolerate conflicts.[12]

A.2.3 Condition variables

ECRAM provides a simple mechanism to avoid busy waiting for object state changes, similar to synchronization with condition variables.[13] The `ecram_wait` call blocks until an object is in a specific state. However, upon returning from the call, an application must check whether the object is still in the requested state. Also, short durations of specific states can remain unnoticed, because, unlike `pthread_cond_wait`, `ecram_wait` is not coupled to a mutex.

[3] config parameter `ECRAM_ENABLE_ACCESS_DETECTION`
[4] config parameter `ECRAM_ENABLE_READ_WRITE`
[5] config parameter `ECRAM_ENABLE_READ_WRITE`
[6] config parameter `ECRAM_ENABLE_MMAP`
[7] config parameter `ECRAM_ENABLE_TRANSPARENT_RESTART`
[8] config parameter `ECRAM_ENABLE_FPU`
[9] config parameter `ECRAM_ENABLE_SYNC`
[10] config parameter `ECRAM_ENABLE_TRANSACTION_INFO`
[11] config parameter `ECRAM_ENABLE_STATISTICS`
[12] config parameter `ECRAM_ENABLE_SKIP_VALIDATION`
[13] config parameter `ECRAM_ENABLE_WAIT`

A.2.4 Nameservice

A simple nameservice has been built into ECRAM (currently only usable with direct-mapped objects).[14] An application can store an object ID under a name using `ecram_nameservice_get`, and retrieve the object ID for a specified name using `ecram_nameservice_set`.

To explore subtrees in the nameservice, the nameservice contains two functions that apply a passed function to several entries in turn. The function `ecram_nameservice_apply` applies the function recursively to the decendants of a specified nameservice entry. Similarly, the function `ecram_nameservice_list` applies the function non-recursively to the children of an entry.

A.2.5 Debug Interface

Each ECRAM module should have a function `module_debug` taking a pointer to a string, i.e. a `char **`.[15] If a string is supplied, it can be parsed to read additional debug parameters. The updated position in the string should be written back.

A.2.6 Unstable Interface

Some functions in ECRAM are declared as unstable, because they do not fit into the clean interface and might be dropped at some point in the future.[16] Examples for such functions are `ecram_set_nodename`, `ecram_is_initial_node` and `ecram_get_own_node_id`. A well-designed application should not rely on these functions to exist or to work as expected.

A.3 Developing Applications

First, we describe the prerequisites to building and using ECRAM. Second, we walk through the process of configuring, building and running an ECRAM application step by step. Third, we introduce several example applications that can serve as starting points for developing applications.

A.3.1 Prerequisites

Before you start with ECRAM, ensure that the following software packages are installed on your system:

- GCC —`gcc`
- Make —`make`
- Libc —`glibc-dev`
- GLib with thread support —`libglib-dev/libgthread-dev >= 2.14`
- readline —`libreadline-dev`
- bfd —`binutils-dev` — only needed for the extended backtrace functionality[17]
- fuse —`libfuse-dev` — only needed for building the FUSE module[18]

The prerequisites will not be a problem on any current Linux distribution. Some distributions have slightly different names for the packages, such as `xyz-devel` instead of `xyz-dev` for development packages.

[14] config parameter `ECRAM_ENABLE_NAMESERVICE`
[15] config parameter `ECRAM_ENABLE_DEBUG`
[16] config parameter `ECRAM_ENABLE_UNSTABLE`
[17] config parameter `ECRAM_ENABLE_EXTENDED_BACKTRACE`
[18] config parameter `APPS_FUSE`

A.3.2 Running ECRAM Applications

The following description helps you build and start up an ECRAM application for the first time.

1. Get the ECRAM source code and change to its top-level directory:
   `cd ~/ecram`
2. Configure ECRAM to suit your needs:
   `make configure`
3. Build the ECRAM library and the provided applications:
   `make`
4. Start the first instance of an application:
   `LD_LIBRARY_PATH=build build/apps/basic/basic -a 127.0.0.1` Setting `LD_LIBRARY_PATH` enables your application to find the ECRAM library without installing it system-wide.
5. On another console, start another instance of an application:
   `LD_LIBRARY_PATH=build build/apps/basic/basic -a 127.0.0.2 -b 127.0.0.1`

As a convention, the `-a` parameter specifies the address to listen for incoming connections, and the `-b` parameter specifies the address of the bootstrap node. Type `q <Enter>` to quit the command shell. After having managed to start two instances of an application on the localhost (127.0.0.x), try to start more instances of the application on different computing nodes.

A.3.3 Understanding Distributed Objects

To get a first understanding of the distributed objects provided by ECRAM, try several commands in the basic application's interactive shell. First, look at the command categories provided by the basic application, and at the commands for object management. Enter the characters after the prompt, and press the *Enter* key.

```
basic >?
basic >?o
```

Second, start a transaction, allocate an object of 20 bytes, register it in the name-service, and end the transaction.

```
basic >tb
BoT
basic >oa20
allocate
allocated 20 bytes at 0x10000d000
basic >ns /hello 0x10000d000
set value for name
/hello <- 0x10000d000
basic >te
EoT
```

Third, switch to the console running the second node and print the name-service entry, outside or inside a transaction.

```
basic >ng /hello
get value for name
/hello -> (nil)
basic >tb
```

```
BoT
basic >ng /hello
get value for name
/hello -> 0x10000d000
basic >te
EoT
```

Note that an access outside a transaction not necessarily retrieves the newest version.

Experiment with waiting for an object condition with command `c=42,0x10000d000` and modifying an object with command `ow0x10000d000,42` (in a transaction). Then try to cause a conflict between concurrent transactions, e.g. start two transactions, write to the same object and finish both transactions. You should observe the second transaction fails to commit and is transparently restarted by ECRAM, i.e. all objects will be restored to their initial state.

Map a file with `fm README` and retrieve file information on the other node with `fi 0x100010000`, passing the ID of the file object. Finally, dump the mapping with `ddd 0x10000f000,607`, where `0x10000f000` is the object ID of the memory-mapped file data and `607` is the size of the mapping.

A.3.4 Example Applications

The `src/apps` subdirectory contains several example applications. The *idle* application contains all code needed to start or join a distributed ECRAM application. The key line in the source code is

```
int ret = ecram_startup(address, port, bootstrap_address,
    bootstrap_port);
```

The *simple* application is an example for allocating objects, using transactions and storing and retrieving entries in the name-service. Look at the function `test_transactional_consistency` to understand how to allocate and initialize an object, store it in the name-service, and busy-wait for another node to modify the object. Once you have figured out how the code works, modify it to use `ecram_wait` instead of busy-waiting.

For a more advanced example on using ECRAM, see the *basic* application. The MapReduce applications such as *wordcount* and *raytracer-mr* are explained in a dedicated section later in this document.

A.4 Objects

Object management comprises the distributed ID space, heaps of objects, object allocation dispatcher, access management and MMU control.

A.4.1 Object Allocation

Objects are allocated using a layered approach. The distributed ID space is partitioned into regions. Each region is assigned to one node. To create smaller objects in regions, regions are handled as heaps.

Distributed ID Space

ECRAM partitions the distributed ID space using regions of objects. The ID space management is implemented in `space.c`. For efficient object lookup, the management will eventually use the key-based routing module `kbr.c` (not implemented yet).

Heap Management

A heap is a region of allocatable objects that are bound to a specific node. The heap module sub-allocates in memory regions obtained from the space module.

Object Allocation

Object allocation requests go to the object allocation dispatcher implemented in `object.c`. The dispatcher decides from which heap to allocate the object. The decision depends on the allocation attributes passed to `ecram_alloc`, but can also be based on monitoring of allocation behaviour or heuristics.

Small objects can be allocated with low space overhead using the *mspaces* allocator in `malloc.c`. The *mspaces* allocator allocates heaps using `object_mmap`. Large objects can be allocated as one entrire heap using the *page* allocator.

The `object_free` function should give back the object to the heap it has been allocated from. The function is currently not implemented, because freeing storage to a remote heap is difficult.

The function `object_mmap` allocates an entire heap and, if a file descriptor is passed, copies the file data into the object. The munmap functions frees the heap, it does not write back the file data.

A.4.2 Object Accesses

Object can be accessed by writing directly to the virtual address corresponding to the object ID (for direct-mapped objects only), or by using read/write functions. Accesses are broken down internally to object blocks.

Block Size

The developer can select the minimum size of an object block.[19]

- Object block size sz $<$ 4KB is only possible for flexible objects. Direct-mapped objects require object block size being a multiple of 4KB.
- Backing storage (physical memory mappings) for direct-mapped objects is created on demand.

Backing Storage

The `access.c` module provides backing storage for direct-mapped objects. These mappings merely cache data from the replication module. Only during transactions that contain direct writes to memory, mappings may contain updated (not yet committed) data. Therefore, we can discard mappings in case of memory pressure.

Linux restricts the size and number of memory mappings (virtual memory areas, VMAs) per process. To save mappings, we support allocating multiple objects in one memory region.[20] Mappings are periodically pruned to save virtual memory using the function `access_prune_mappings`.[21]

Accesses Via Read and Write Functions

All accesses go through the call dispatcher `dispatcher.c`. For direct-mapped objects, the functions `dispatcher_read` and `dispatcher_write` translate OID and offset to block alignment in ECRAM, because OID or offset might exceed `ECRAM_BLOCK_SIZE`. Flexible objects are not implemented yet here.

The functions `read_aligned` and `write_aligned` assume translated OID, offset and size, i.e. OID aligned to `ECRAM_BLOCK_SIZE` and offset plus size less than or equal to `ECRAM_BLOCK_SIZE`. First they retrieve the preferred version of the accessed object from the consistency module. Second, they perform the access by calling `access_read` or `access_write`. Third, they register the access with the consistency module.

[19] config parameter `ECRAM_MINIMUM_LOG2_BLOCK_SIZE`
[20] config parameter `ECRAM_NUM_BLOCKS_PER_MMAP`
[21] config parameter `ECRAM_MAX_MAPPING_MEMORY`

The access_read and access_write functions first prepare the access with access_prepare and then transfer data from or to the replication module with access_export or access_import if a buffer is supplied and the version defined. For flexible objects, data must be transferred directly between input/output buffer and replication module (not implemented yet).

For direct-mapped objects, preparing an access means creating a mapping, loading the replica into the mapping and granting access using the MMU module. Similarly, finishing an access with access_finish revokes the access privilege using the MMU module. In case of write accesses, an object may be modified between access_prepare and access_finish. The version of modifiable objects is set to undefined_replica_version to ensure that fresh data will be loaded when access_prepare is called again.

The access_reinit function is called after restructuring of region allocation. It resets the object to its default state, i.e. zero-filled content.

MMU-based Access Detection

The mmu.c module interfaces between MMU-based memory access detection and the dispatcher for read and write functions dispatcher_read and dispatcher_write. It does not store any state by itself. To catch page faults, the module installs the segfault_handler signal handler for SIGSEGV. The functions mmu_prepare_read, mmu_prepare_write and mmu_finish allow to change the access rights of the memory page specified by the OID. They are typically invoked by the access module to allow accesses, or by the rc module to request access detection.

Inter-module Call Structure

Figure A.1 presents the inter-module function call hierarchy for object accesses. The library interface read/write functions as well as the segfault handler for MMU-based accesses call the function call dispatcher. MMU-based accesses do not call access_export or access_import, because there is no buffer to transfer data.

A.4.3 Naming Objects

To enable a distributed application to anchor its data structures, the root module defines a set of root objects. Using root_set, an application can register an object ID under an application-defined index. A root object's ID can be retrieved using root_get. The module defines at least 256 entries for root objects. The index ROOT_WORLD is already predefined, and ROOT_NAMESERVICE is the anchor of ECRAM's built-in name-service.

The built-in name-service is implemented by the nameservice module. The functions match their counterparts from the ECRAM interface. The name-service has some predefined entries for the root objects described above, such as /world and /nameservice. Nameservice entries are organized hierarchically, however, each entry stores at most his first child and one sibling for chaining entries at the same level.

A.5 Replication

The replication module stores the permanent data content of objects. It is also responsible for notification about object changes.

A.5.1 Versions and Replicas

Object versions are specified using the combination of OID and version number (replica_version_t). The replication service is neutral to version numbers, except for undefined_replica_version, which acts as a negative result or wildcard, and

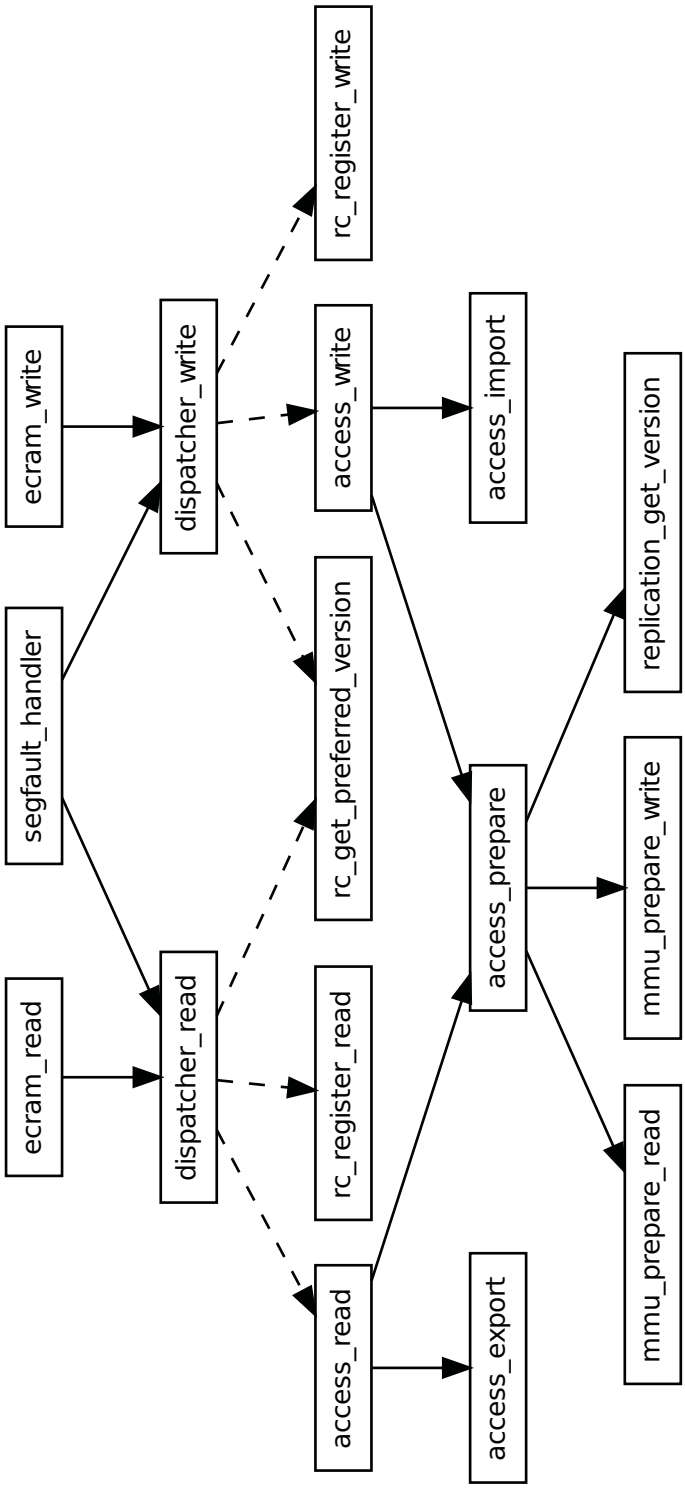


Figure A.1: Call graph for object accesses

`initial_replica_version`, which can always be reconstructed in a local operation. A higher version number is considered newer. However, the consistency module defines version IDs and their compatibility.

An object version that exists at a specific node is called a replica. Besides the payload data and the version number, the replica structure can store the previous version number, the version number which invalidated this version, the node who produced this version, and the object's size, which might vary between different object versions. Fields in the structure that are unknown may remain undefined. For example, if the data is currently not available locally, a replica placeholder can be created with `NULL` data to store the `from` node and request the data later.

Initially, the whole ID space is zero-filled. Also, each allocation of a heap restores the objects in the heap to their initial state. The content of zero-filled versions is encoded as `ZERO_FILLED`.

A.5.2 Module Interface

The interface functions of the replication module consists of the functions `replication_create_version`, `replication_get_version` and `replication_wait`. The function `replication_create_version` to create or update a version created either by the node itself or by a peer node. The complementary function `replication_get_version` writes the specified replica's data to a memory buffer.

A.5.3 Version Comparison

The module has two different comparator functions: The `replica_version_compare` compares not only versions but also invalidation versions. When searching for `undefined_replica_version` with `replica_version_compare`, any replica that is still valid will do. In contrast, the `replica_version_compare_data` is more exact: It does not accept the `undefined_replica_version` wildcard, and it does not look at invalidation versions.

A.5.4 Replica Access

The low-level function `lookup_or_create` retrieves or creates a replica. The version parameter specifies which replica is requested. The constant `undefined_replica_version` acts as a wildcard for the highest known version number. Looking up `undefined_replica_version` may create `initial_replica_version` if nothing else known about the version. The `fuzzy` parameter specifies whether a replica with an older version that seems to be valid through version may be returned.

The function `create_or_update_version` works at a higher level. It invalidates any previous version, ensures the replica structure exists using `lookup_or_create` and stores the payload data and other fields in the structure, potentially overwriting values that were undefined so far. Finally, the function checks whether the local node was waiting for a state change for this object. Replica updates during local commits are possible by specifying `undefined_replica_version`.

The function `get_version_from_remote` creates a replica by retrieving the version from a peer node. If the manager node for this version is unknown, it asks the space module for the probable manager. If every other attempt fails, the node contacts its bootstrap node.

A.6 Consistency

All consistency related library calls go through the ecram module and the dispatcher module. Transactional consistency and different variations thereof are implemented in the rc module.

A.6.1 Call Dispatcher

The consistency dispatcher forwards function calls from the ecram module to the responsible module. It translates function call arguments such as object IDs and attributes to the semantics required by the module.

For transaction management, the dispatcher implements flat nested transactions, transactions may occur inside transactions, but all accesses are attributed to the outermost transaction, which is the only transaction passed through to the rc module. In the `dispatcher_eot` function, `access_prune_mappings` is called allow the access module to save memory by removing old memory mappings.

For direct-mapped objects, the object ID, offset and size parameters to read and write calls are adapted to the block alignment.

A.6.2 Speculative Execution

The remainder of this section is implemented in the rc module. During speculative execution of a transaction, all accesses are recorded in the `accessed_objects` structure. The information stored is the object ID, size, version accessed (previous), and the type of access. If the access is a write, the (current) version field is set to undefined, because it will become defined after validation of the transaction. For a read access, the version field equals the previous field. The rc module also tracks allocations and frees in order to be able to revert them in case of a transaction failure.

After entering the `rc_eot` function to end a transaction, the information gathered during speculative execution is transformed into a `transaction_t` structure by `build_transaction`.

A.6.3 Transaction Information

To be able to validate transactions, the rc module stores each object's top-most version number in the `versions` hash-table. It also keeps a history of recent transactions, which serves to update the object version numbers in sequence without omitting or reverting an object. The updating of object versions is done by `update_versions`. The function `insert_transaction` function imports a transaction to the history and object versions.

A.6.4 Transaction Validation

Transaction validation is currently implemented via a central validator node. A non-validator node offloads transaction validation to the validator by calling `remote_validate_and_commit`, which sends the transaction as a `rc_validate` message to the validator. If the validator finds the transaction to be valid, the originating node receives a defined version number for the transaction, which it stores in the transaction structure.

If the validator node receives a `rc_validate` request, it calls `validate_and_commit` and replies with either the valid transaction's defined version or with `undefined_replica_version`.

Nodes that are not involved in a specific transaction will be notified of it by means of a `rc_commit_notification` message. The notification handler creates replicas or placeholders for the objects modified by the transaction. It also inserts the transaction into the transaction history.

The low-level validation is implemented in the `validate` function. Validation can only run if all transactions are known and contained in the history. Therefore, the validation function waits for the global transaction version `top_version` to equal the version until which the transaction history is complete (`complete_version`). Then the function checks for each object in the transaction's read and write set whether the previous version still equals the current version known for the object from the `versions` table. Any object that has been updated during speculative execution causes the transaction to be invalid. The optimizations for read-only transactions are optional.[22]

[22] config parameter `ECRAM_ENABLE_READONLY_TRANSACTIONS`

A.6.5 Local Commits

Local commits can update an object without global validation. They can take place only if all objects accessed by the validating transaction have not been replicated. Therefore, the `validate_and_commit` function needs to ensure that no replicas are handed out during local validation and commit (`local_commit`) using `replication_disable` and `replication_enable`. This is severe inter-module locking and may be considered bad.

A.7 Messaging

ECRAM's communication subsystem consists of TCP-based networking, node management and messaging. Furthermore, a `node_info_block_t` represents each node as an object. The key-based routing module for sending messages in a DHT-like manner over the network is not yet functional.

A.7.1 Networking

The networking module `net.c` stores connections in two hashtables, one indexed with Node-IDs, the other indexed with socket numbers.

Data is sent over the network with the `net_send` function. Its `message_t` parameter contains all information needed: to which node to send the message, the payload data, the length of the payload etc.

To receive messages from other peers, the module starts a network handler thread. The thread runs an endless loop, blocking on `epoll_wait` until the epoll mechanism signals pending events. For an incoming connection request, the event's socket is the `myself.socket`, which results in a call to `epoll_accept_connection` to identify this node by sending a hello message. The `EPOLLOUT` flag signals that a connection has been established, in which case `epoll_established_connection` is called. If a message has been received, the `EPOLLIN` flag has been set, and `epoll_receive` is called.

The `epoll_receive` function prepares the peer's receive buffer and reads data from the TCP stream into the buffer with `receive_data` in non-blocking mode. If everything went well until now, `decode_message` extracts ECRAM messages from the buffer. This function will in turn transfer control to `message_handle` in the message module. Finally, `compactify_buffer` is called to ensure that subsequent receive operations will not exceed the buffer's capacity despite partial messages remaining in the buffer.

During bootstrap, the function `net_update_id` enables changing the ID of oneself and of the bootstrap node.

A.7.2 Node Management

The `node.c` module contains the functionality to join the network by requesting a node ID from a bootstrap node. It also allows to request connection information about third-party nodes. While bootstrapping, a node uses the `undefined_node_id`, such that the reply to a bootstrap request must identify the joining node by the socket it is connected to.

A.7.3 Sending and Receiving Messages

Messages are classified using a type and a subtype. Typically the type corresponds to the module that sends the message, and the subtype is internally defined by the module.

Receiving

The message module handles incoming messages in the network thread in function `message_handle`. This function looks up the module that will handle the message. A return

value of 0 means that the message structure can be deleted by the caller, a return value of 1 means that another thread will free the message, because the message is a reply that has been attached to the corresponding request message, which is identified using the `in-reply-to` field.

A reply message is passed to `process_reply` and in turn to the specified module's reply handler. The reply handler is called with the original message as argument, such that the reply can be found in the message's `reply` field.

Sending

There are several slightly different functions for sending messages:

- The `message_send` function sends an asynchronous, i.e. one-way and non-blocking, message.
- The `message_send_sync` function send a synchronous message, which blocks until the corresponding reply has been received. To deal with node failures, the function should be extended with a timeout mechanism and error handling.
- The `message_reply` function sends a reply to a specified request message. Sending a reply is non-blocking.
- The `message_multicast` function sends a message to a list of peers. The current implementation assumes that multicast messages are one-way. Otherwise, the reply processing needs to be extended to work with multiple replies to one message.

A.8 Debugging and Monitoring

The facilities described in this section assist the developers in improving ECRAM.

A.8.1 Debugging

ECRAM's debug output depends on the global debug level[23] and on the per-module debug levels, whatever value is higher. The default debug level is zero, which disables most debug output, but keeps the code compiled in. The debug level can be changed during run-time by calling `set_debug_level_<modulename>` which is an assembler alias to the `set_debug_level` function. Severe errors are output unless the debug level is set to -1.

Debug output is produced using the `dbg_printf, dbg_warn, TODO, dbg_perror and PANIC` macros. The first macro takes the minimum debug level when to print the output, the other macros print the output unconditionally.

ECRAM developers should catch all potential error cases by placing assertions in the code. The `ASSERT(expr)` macro evaluates the argument and, if non-zero, prints on the debug output that the assertion does not hold.

The function `debug_dump_config` prints the config data which is embedded in the ecram library's binary. The function `debug_dump_memory` prints the content of the specified memory in hexadecimal and string format.

A.8.2 Monitoring

The monitoring service is designed to be minimally intrusive: It can be turned off completely.[24] The `monitor.h` header file avoids naming collisions by prefixing all symbols with `monitor_`.

Monitoring in ECRAM works by marking entities in the source code. Every time the code reaches the entity, a monitoring event is generated, which causes a handler function to be called. For

[23] config parameter `GLOBAL_DEBUG_LEVEL`
[24] config parameter `ECRAM_ENABLE_MONITORING`

each entity to be monitored, the monitoring subsystem adds control information to a module's data (`monitor_control_t`).

To monitor an entity, declare it using `MONITOR_DECLARE_EVENT(entity, handler)` or using `MONITOR(entity)` if the `ECRAM_MONITOR_entity` has been declared in the configuration. Use `monitor_trace_event(entity, user_data)` etc. to weave monitoring events in the source code. Alternatively, `monitor_begin_event` and `monitor_end_event` allow to record the entry and exit into a piece of code. The function `monitor_dump_all` causes all monitor entities to be printed by their specific handlers. The developer can modify handler functions during run-time by calling `monitor_set_handler(char *control, char *handler)`. The file `handler.c` defines various handler functions for immediate output, time measurements, collecting user-supplied data and printing call backtraces.

A.8.3 Wireshark Packet Dissector

The Wireshark network protocol analyzer provides a graphical frontend to record, sort and filter network traffic. As described above, ECRAM messages have a fixed-size PDU header that contains the overall length of the packet. ECRAM network traffic usually comes from or goes to ECRAM's default IP port 2001.

Starting up the ECRAM dissector in `plugin_register` registers two structures: The `hf_register_info hf` registered using `proto_register_field_array` describes the primitive data fields in the ECRAM protocol. The `gint *ett[]` array registered using `proto_register_subtree_array` holds the expansion states of the subtrees.

The dissection of ECRAM packets starts in the function `dissect_ecram`, which reassembles message fragments from the TCP data stream, because data chunks received from sockets need not correspond to ECRAM messages. On each ECRAM message found in the TCP stream, Wireshark calls the function `dissect_ecram_message`, which takes as arguments the `tvbuff_t *tvb` containing the message data, the `packet_info *pinfo` describing what to display, and the root `proto_tree *tree` of the protocol tree to build. First, the dissector function extracts the elements of the message header and inserts them into the tree. Then it extracts and inserts the specific payload data depending on the message's type and subtype.

A.9 DTK – Job Management

The job management can be used to let nodes execute custom job functions. The node that assigns the jobs is the master, and the other nodes take the role of workers. The communication between the master and the workers is based on job queues. Depending on the preprocessor definitions there may be one global job queue or many private job queues, one for each worker. In general, the master just needs to add a job to a job queue (global or private) to make sure, that it is taken care of. As long as there are jobs in the job queue a worker continues to get jobs from the queue and executes them. Once the job queue is empty, the worker remains in a standby state, waiting for new jobs to arrive by using an `ecram_wait` call on the number of jobs in the job queue. Depending on the preprocessor definitions, a worker may try to steal a job from another job queue before he enters the standby state.

A.9.1 Preprocessor definitions

It is possible to *run mapreduce jobs from private queues for each worker instead of a global queue.*[25] If this option is enabled, it is also possible to *enable job stealing from local job queues.*[26]

[25] config parameter `MAPREDUCE_RUN_LOCAL`
[26] config parameter `MAPREDUCE_JOB_STEALING`

A.9.2 Interface functions

`job_startup` function

The `job_startup` function initializes the job module. Both the master node and the worker nodes have to call this function at the beginning. The first node that calls this function creates the global worker queue and the global job queue. Then it registers both with the nameservice. All other nodes retrieve these queues from the nameservice.

`job_wait_for_workers` function

This function can be used by the master to wait for a certain number of workers to be online before submitting job functions.

`job_submit` function

The purpose of this function is to assign a job to the workers. If no job queue is specified the job queue gets choosen internal. In case of a global job queue all jobs are added to this queue and the FCFS policy is used, to assign the jobs to the workers. In case of private job queues, a round robin scheduling is used to distribute the jobs evenly among the workers. The job object can be set over the parameters of the function. The movable variable determins if the job may be stolen by another worker. The completion variable can be used by the master to check, if the job has finished. So it is possible to assign a group of jobs with the same completion variable and then check for the whole group of jobs, if it has finished (see code example).

`job_run` function

This function is the entry point for the workers. First, the worker registers himself in the global worker queue with help of the `register_worker` function. Then the worker fetches the job queue (`get_job_queue` function). Next the worker begins with the execution of the job functions (`job_loop` function). If the job queue is empty, the worker waits until a job is added to the queue (`wait_for_job` function). The worker continues executing and waiting for jobs until he receives a “terminate” function from the job queue. Then the worker unregisters himself (`unregister_worker` function).

`job_terminate` function

This function adds a "terminate" function to the job queue.

`job_terminate_all` function

This function calls the `job_terminate` function n times, where n is the number of registered workers.

`job_get_workers` function

This function returns the number of registered workers.

A.9.3 Internal functions

`register_worker` function

The worker allocates and sets a `worker_node_t` object and adds it to the global worker queue.

`get_job_queue` function

This function returns the global or the local job queue, depending on the preprocessor definition.

`job_loop` function

This is the main function of the job module, where workers loop waiting for jobs and running them until the terminate variable in the `worker_node_t` object is set to 1. Basically, in one iteration, the functions `wait_for_job`, `get_job`, `run_job` and `finish_job` are called, but there is also some logic for the job stealing at the begin of the function: The function `idle_nexttime` gets called to check if there are jobs left in the current job queue which wait for execution. If there aren't any jobs left, the function `steal_work` gets called to steal jobs from other job queues.

`wait_for_job` function

The function makes an `ecram_wait` call with the condition `jobs->nwaiting != 0`, which means that there are jobs in the queue waiting for execution.

`get_job` function

This function moves a job from the inner job waiting queue to the inner job processing queue of a job queue and sets the process flag of the job to a given worker. There is also some logic for the job stealing in this function: The `steal_work` function calls the `get_job` function to steal a job from another worker. There are some jobs that may not be stolen, e.g. a finish job. These jobs have the "movable" flag set to 0. If a worker tries to steal such a job with help of the `get_job` function, the function will return `undefined_object_id`.

`run_job` function

The `run_job` function runs a job by calling one of the custom job functions. These functions need to be declared in a job function object. In case of a terminate function the terminate flag of the worker is set to 1 and the function returns.

`finish_job` function

This function removes a finished job from the inner job procssing queue of the job queue.

`idle_nexttime` function

This function checks whether there are jobs left in a given job queue. If there are'nt any jobs left there are two options: If job stealing is disabled, the worker gets blocked until new jobs have arrived. Elsewise the `steal_work` function gets called to steal a job from another job queue.

`steal_work` function

First, the function checks if there are jobs in the global job queue. If this is the case the `get_job` function gets called for the global job queue. If there is no job in the global job queue the function randomly determins a worker and checks his job queue for waiting jobs. If there are no jobs waiting, the function repeats the last to steps for a maximum of n times, where n is the number of registered workers. If a non-empty job queue was found, the `get_job` function for this job queue gets called.

A.9.4 Data structures

`job` struct

The `job` struct contains all information of a job:

- the name of the function to execute
- the input of the job
- the output of the job
- the variable movable which specifies if the job may be stolen by another worker
- the variable completion which is set after the job has finished
- the worker, which executes the job
- the timestamp of the start of the execution

`job_queue` struct

Depending on the preprocessor definition there is a global job queue for all jobs or each worker has it's own job queue. A `job_queue` stores the following information:

- the total number of jobs in the queue
- a queue for the jobs, which wait for execution
- a queue for the jobs, which are executed at the moment
- the number of the jobs, which are waiting
- the number of the jobs, which are executed at the moment

`worker_node` struct

The `worker_node` struct contains the following information of a worker:

- the variable terminate, which is set if the worker should terminate
- a pointer to the local job queue of the worker
- the id of the worker
- the variable starting which is set to 1 during the starting phase
- the variable is_thief, which is set to 1 while the worker is stealing jobs from other workers
- a pointer to the other node's queue, where jobs get stolen from

`worker_queue` struct

The `worker_queue` contains the following information:

- the number of workers in the queue
- the variable `starting_phase`, which also contains the number of workers in the queue and is used for the distribution of jobs to the job queues in the starting phase, if `private_queues` are enabled (see preprocessor definitions).
- the queue of the workers

`job_function` struct

The `job_function` object contains the following information:

- the name of the function
- a function pointer to a custom job function

A.9.5 Debug functions

With help of the debug functions it is possible to print information about a job (`job_debug_job` function), to print information about a job queue (`job_debug_queue` function), to print information about a worker (`job_debug_worker` function) or to print all these information (`job_info` function).

A.9.6 Code example

```
/*the job function object */

job_function_t example_functions[] =
{
        { "example_map", example_map },
        { "example_reduce", example_reduce },
        JOB_END_OF_FUNCTIONS
};

job_startup(ecram_is_initial_node());

if (ecram_is_initial_node())
{
  ...

  /*wait for nworkers to be online*/

  job_wait_for_workers(nworkers);

  ...

  /*submit a group of jobs with the same specific custom job function
      (app->map_function="example_map", queue=ecram_undefined_object_id)*/

  for (job = 0; job < nmaps; job ++)
  {
    job_submit(queue, input_split, app->map_function, intermediate,
        &app->ncompleted_maps, 1);
  }

  /*wait for this group of jobs to be finished*/

  ecram_wait(&app->ncompleted_maps, 0, ecram_wait_equal, nmaps);

  ...

  /*submit another group of jobs with the same specific custom job function
      app->reduce_function="example_reduce",
      queue=ecram_undefined_object_id*/
```

```
  for (job = 0; job < nreduces; job ++)
  {
    job_submit(queue, reduce_input, app->reduce_function, final_result,
        &app->ncompleted_reduces, 1);
  }

  /*wait for this group of jobs to be finished*/

  ecram_wait(&app->ncompleted_reduces, 0, ecram_wait_equal, nreduces);

  /*terminate*/

  job_terminate_all();

}
else
{
  /*entry point for workers*/

  job_run(job_functions, 0); //0: use local queues, 1: use the global queue
}
```

A.10 DTK – MapReduce

A.10.1 MapReduce

MapReduce is a computing model which has been suggested by the Google employees Dean and Ghemawat in 2004. MapReduce restricts the execution flow and data access of applications to achieve a high degree of parallelism. An application that adheres to the MapReduce model consists of two phases: The map phase splits input data such that several worker nodes can compute intermediate results in parallel. The reduce phase transforms the intermediate results into the final result, again in parallel. A dedicated master node splits, shuffles and merges data and assigns jobs to worker nodes. In the original MapReduce model, data dependencies occur only between input and intermediate data respectively between intermediate and output data, such that both phases are embarrassingly parallel, which means that in the map and the reduce phase there is nearly no communication between workers necessary. Thus, MapReduce simplifies synchronization at the expense of restraining data dependencies and control flow.

A.10.2 ECRAM MapReduce Framework

The ECRAM MapReduce Framework is an in-memory, extended MapReduce framework. The framework stores shared input, output and intermediate data in ECRAM. This enables data orientated communication. Also the framework itself stores data in ECRAM. The framework also supports iterative and on-line data processing.

A.10.3 Framework

Interface

The interface is defined in `lib/dtk/ecram.h`. The `mapreduce_run` function is the entry point of the mapreduce framework. As parameters it takes the name of the application, the master function and a pointer to the job functions. It acts as an dispatcher. The master function is assigned to the initial node, which takes the roll of the master. All other nodes become workers and take care of the job

functions. After configurating the mapreduce framework, the master function calls the `mapreduce` function and the application starts.

User Defined Functions

Only the master function is obligatory. All other functions are optional. The master, pre, post, shuffle functions are executed on the master. All other functions are executed by workers. Depending on the number of iterations, all functions with the exeption of the master function are repeated several times. The chronological order of the functions is equivalent to the order in this document.

master function All of the configuration takes place in the master function with help of the `mapreduce_application_t app` object. Also parsing of user-defined input data can be implemented in the master function. Therefore the `mapreduce_storage_t app->input` object can be used.

pre function The `pre_function` takes place after preparing the input data and before splitting it. It has access to the `app->config` object, the `input->data` object and the variables `input->length` and `iteration`. Pre-processing of the input data before each iteration is the purpose of this function.

prepare map functions After splitting the input and preparing storage for the intermediate results (results of the map functions), the prepare map functions are called. The number of prepare map functions is identical to the number of map functions. These functions have also access to the same objects as the map functions. These are a `mapreduce_storage_t` object and an intermediate result. The first contains several information of the input data and also some meta information to split the input and the latter is a pointer to a block of allocated memory.

map functions The map functions usually process the input data and save the intermediate results, so that the reduce functions can use these results to get the final results. Before the map functions are called, the input data gets splitted into equal pieces, one for each map function. Therefore offset and length are calculated, stored in an `mapreduce_storage_t` object, which contains also a reference where to find the input data, and then passed to the map functions. Because there are no dependencies, each map function can process its input split independently. However, the results need to be merged by the reduce functions to gain the final results. To save the intermediate results from the map functions, a block of allocated memory is divided into equal pieces, one for each map function. A pointer to this block is passed to the reduce functions, so that they have access to all intermediate results.

shuffle function The shuffle function can be used to prepare the intermediate results for the prepare reduce and reduce functions after the map jobs have finished. It therefore has access to all intermediate results.

prepare reduce functions After preparing the final results, the prepare reduce functions are called. Like the prepare map functions the number of prepare reduce functions equals the number of reduce functions. They also have access to the same objects than the reduce functions, which are a `mapreduce_reduce_input_t` object which contains a reference to all intermediate results and a pointer to the final results. The functions are used for prefetching, preparing statistics etc..

reduce functions The reduce functions process the intermediate results to gain the final results. Each reduce function usually processes only a part from each intermediate result. The `mapreduce_reduce_input t` object contains a reference to all intermediate results and also some meta information to determine which part of the intermediate results is of interest. The functions also have access to a pointer to the final results.

post function After the reduce phase has finished, the post function is called, which has access to the `app->config` object and also the final results, so that it can post-process them after each iteration.

Objects

`mapreduce_application_t app` object The `mapreduce_application_t app` object represents a map reduce application. In the master function all of the configuration can be done using the `app` object. In general the configuration settings are optional. If a configuration parameter is not specified the default value is used instead. As a result of this the developer needs only to take care of the configuration parameters which are in his interest.

functions The functions can be set with the variables `app->pre_function`, `app->prepare_map_function`, `app->map_function`, `app->shuffle_function`, `app->prepare_reduce_function`, `app->reduce_function` and `app->post_function`. Note that the functions which are executed by workers need to be defined in an `job_function_t` object which is a parameter of the `mapreduce_run` function. Only the name of these functions is then assigned to the `app` object. The other function variables in the `app->object` are direct function pointers. Only the functions which are defined in the `app` object are executed.

number of iterations There are three variables which control the number of iterations: `app->max_iterations`, `app->iteration` and `app->iterate`. Usually the number of iterations is set in `app->max_iterations`, after each iteration the `app->iteration` variable is increased and as long as `app->max_iterations > app->iteration` the iteration continues. A second condition for continuing the iteration is `app->iterate != 0`, but if the `app->max_iterations` is set to a value greater than 0, `app->iterate` is automatically overwritten with 1, even if another value is defined in the master function. A local variable in the `pre_function` ist set to the value of `app->iteration`, so that the actual number of iterations can be seen. However, in the `reduce functions` there is acces to the `app->iteration` and `app->iterate` variable, so that the usual behaviour of the iterations can be influenced. If for example the `app->iterate` variable is set to 0, the iteration stops. Figure A.2 presents the flowchart for MapReduce iterations.

input The preparation of the input data depends on the `app->input` object and the `app->input_descriptor` variable. An input file can be mapped automatically with the map reduce framework, if the `app->input` object is null and an input descriptor is specified in `app->input_descriptor`. The `prepare_map` and `map` functions then have access to the input data with the `ecram_file_t mapping`, `ecram_object_id_t data` and `size_t length` variables within the `maprecuce_storage_t` object. For a user-defined input the `mapreduce_storage_t app->input` object can be set and no automatic mapping will occur. The access to the user-defined input in the `prepare` map and `map` functions remains the same. In case of an automatic mapping of the input data the internal variable `input->length` is set to the length of the file. In case no input descriptor is specified and the `app->input` object is null, it is set to 0. In these cases it can be overwritten with the `app->input_length` variable. In case of a user-defined input the variable is defined in the `mapreduce_storage_t app->input` object. This may have some consequences for the later split phase.

splits (number of map jobs) The `app->split_size` variable determines the size of one split and also the number of splits. If not defined, the number of splits can be directly set with the `app->nsplits` variable and the split size is calculated automatically by dividing the `input->length` variable by the `app->nsplits` variable. Otherwise, the number of splits is calculated by the formula `(input->length + split size - 1)/ split size`. For each split

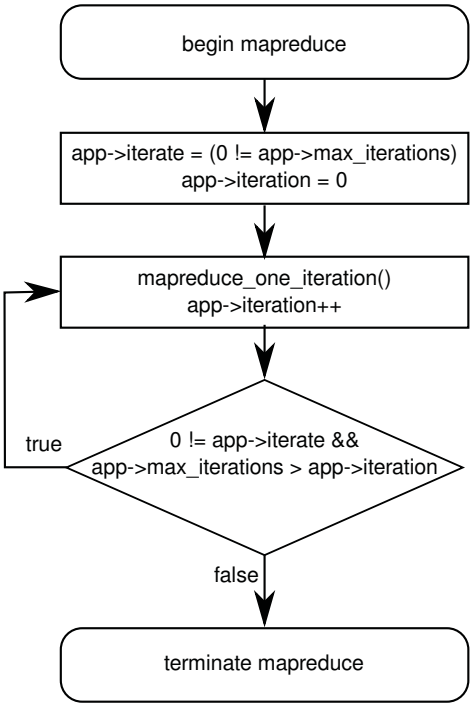


Figure A.2: Flowchart for MapReduce iterations

an offset and a length are set. The length is for all other than the last split the split size. The length of the last split may be shorter if the division of the input length by the split size doesn't come out even. Both values can be accessed in the map phase with the `mapreduce_storage_t` object. Also the variables `npartitions` and `id` within the `mapreduce_storage_t` object are set in the split phase. The first contains the number of splits and the second the number of the map job. These variables can be used to determine offset and length within the map phase, if for example a user defined input is used. If neither `app->nsplits` nor `app->split_size` are specified, `app->nsplits` is set to `app->nworkers * MAPREDUCE_CHUNK_FACTOR`. The default value of `app->nworkers` is the number of worker nodes.

intermediate size (size of memory for the intermediate results of the map phase) The calculation of the intermediate size depends on the variable `app->total_intermediate_size`. If specified, the intermediate size is calculated by dividing `app->total_intermediate_size` by the number of map jobs. Otherwise, it can directly be set in the `app->intermediate_size` variable. If neither `app->total_intermediate_size` nor `app->intermediate_size` are specified, `app->intermediate_size` is set to the default value `MAPREDUCE_INTERMEDIATE_SIZE`.

number of reduce jobs The number of reduce jobs can be determined with the variable `app->nreduces`. If not specified, there will be no reduce job.

number and size of final results The size of memory for the final results can be determined with `app->final_size`. If not specified, the size is set to `app->nfinals * sizeof(ecram_object_id_t)`.

storing the output If the `app->output_descriptor` is specified, the final results are written to a file after each iteration.

`mapreduce_storage_t` objects The `mapreduce_storage_t` objects are the input objects for the map phase. They contain the following information: The objects `mapping` and `data` which are set during the preparation of the input and described in the earlier *input* paragraph. The variables `offset, length, id` and `npartitions` which are set in the split phase and described in the earlier *split* parapgraph. The `config` object which can be set in the master function with the `app->object` parameter. Each map job has its own offset, id and also the length may differ whereas the mapping, data and config parameters are references to the same objects respectively.

`mapreduce_reduce_input_t` objects The `mapreduce_reduce_input_t` objects are the input objects for the reduce phase. They contain the following information: All intermediate results can be accessed with the `intermediate_results` reference. `intermediate_size` is a copy of `app->intermediate_size` as described in the earlier *intermediate size* parapgraph. `final_size` is a copy of `app->final_size` as described in the earlier *number and size of final results* paragraph. `nintermediates` and `nreduces` are identical and a copy of `app->nreduces` which is specified in the master function. `id` is the number of the reduce job. The two pointers `iteration` and `iterate` are a reference to `app->iteration` and `app->iterate` as described in the *number of iterations* paragraph and can influence the behaviour of the iterations. `config` is a reference to the `app->config` object, which can be specified in the master function. The last three variables are only accessible in the reduce functions and not in the prepare reduce functions.

A.10.4 Preprocessor definitions

The size of the intermediate results can be specified in the config option *size of intermediate data block in MapReduce*[27]. However, if the `app->total_intermediate_size` is specified, it has no influence at all. If monitoring is enabled, it is possible to enable or disable monitoring of the job functions with help of the config option *enable monitoring of transactions and conflicts in map and reduce functions*[28]. There is the possability to *run mapreduce jobs from private queues for each worker instead of a global queue*[29]. If this is enabled it is possible to set the config option *enable job stealing from local job queues*[30]. The config option: *factor by which the number of workers is multiplied to get the number of map/reduce jobs*[31] is only relevant, if neither `app->nsplits` nor `app->split_size` are specified, as described earlier in the *splits* paragraph. The number of map/prepare map jobs is then determined with the formula `app->nworkers * MAPREDUCE_CHUNK_FAKTOR`.

[27] config parameter `MAPREDUCE_INTERMEDIATE_SIZE`
[28] config parameter `MAPREDUCE_ENABLE_MONITORING`
[29] config parameter `MAPREDECE_RUN_LOCAL`
[30] config parameter `MAPREDUCE_JOB_STEALING`
[31] config parameter `MAPREDUCE_CHUNK_FACTOR`

A.10.5 Code example

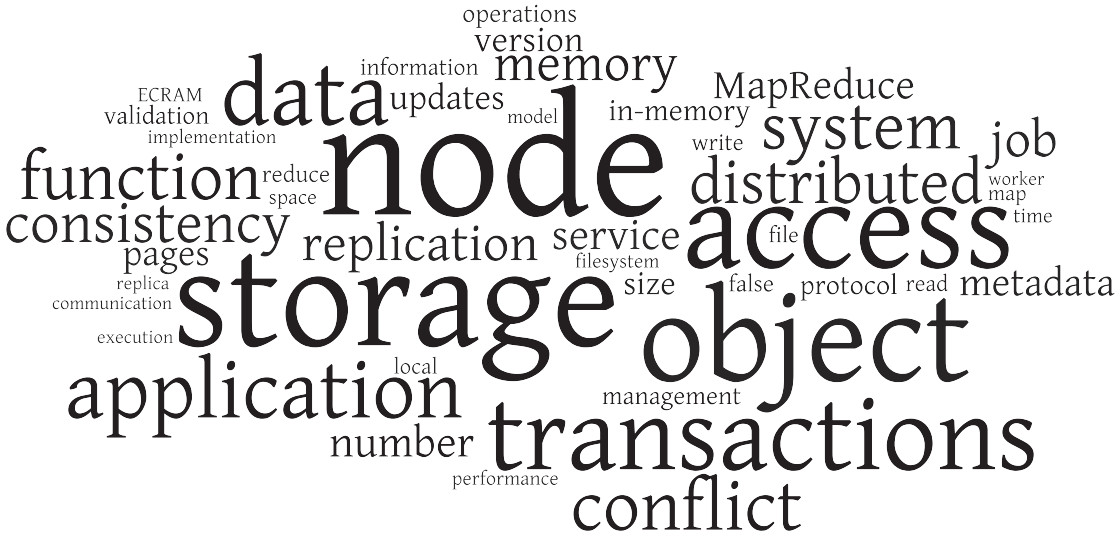
The following code example is copied from the file `/apps/mapreduce/maprreduce.c`. It can be used as a template for developing MapReduce applications.

```
/**
 * example map function
 * @param input          mapreduce_storage_t input_split
 * @param output         reference to ?
 */
void example_map(ecram_object_id_t input, ecram_object_id_t output)
{
        printf(">> example_map input "PRIo" output "PRIo"\n", input, output);
        ecram_bot(0, NULL);
        mapreduce_storage_t *storage = (mapreduce_storage_t *)input;
        ecram_object_id_t *intermediate = (ecram_object_id_t *)output;
        // allocate and register intermediate data structure
        *intermediate = storage;
        ecram_eot(0);
        sleep(5);
}

/**
 * example reduce function
 * @param input          array of intermediate results
 * @param output         reference to ?
 */
void example_reduce(ecram_object_id_t input, ecram_object_id_t output)
{
        printf(">> example_reduce input "PRIo" output "PRIo"\n", input,
            output);
        sleep(5);
}
```

This is the master function, which configures the MapReduce Framework using the app object.

```
/**
 * configure and run mapreduce on master
 */

void example(ecram_object_id_t infile, ecram_object_id_t outfile)
{
        assert(NULL != infile);
        assert(NULL != outfile);
        // create and configure application description
        ecram_bot(0, NULL);
        mapreduce_application_t *app =
            ecram_alloc(sizeof(mapreduce_application_t), NULL);
        memset(app, 0, sizeof(mapreduce_application_t));
        strcpy(app->application, "wordcount");
        app->shuffle_function = NULL;
        app->nworkers = job_get_workers(); // NOTE initialize with
            approximate/current number of workers
        app->split_size = 0; // NOTE example value
        strcpy(app->map_function, "example_map");
        strcpy(app->reduce_function, "example_reduce");
        app->ncompleted_maps = 0;
        app->ncompleted_reduces = 0;
        ecram_eot(0);
        // run mapreduce
        mapreduce(app);
}
```

```
/**
*  declare job functions
*/

job_function_t example_functions[] =
{
        { "example_map", example_map },
        { "example_reduce", example_reduce },
        JOB_END_OF_FUNCTIONS
};

const char *appname = "example";

/**
 * main function of mapreduce application
 */
int main(int argc, char *argv[])
{
        mapreduce_run(argc, argv, appname, example, example_functions);
        exit(EXIT_SUCCESS);
}
```



The tag cloud has been created using Wordle.
This thesis has been typeset using LaTeX, the Gentium font created by SIL International,
the programs Inkscape, Graphviz and Octave.
Figure 6.1 shows the palace *Schloss Benrath* in the south of Düsseldorf. Schloss Benrath was erected under the supervision of Nicolas de Pigage for the Elector Palatine Charles Theodore, a contemporary of the Prussian king Frederick II.