RECODE: Reconfigurable, Consistent and Decentralized Data Services

Mikael Högqvist
Peerialism AB
mikael@peerialism.com

Alexander Reinefeld
Zuse Institute Berlin
ar@zib.de

***Abstract*—Key-based routing schemes, where a message is forwarded towards a server responsible for a partition in a large name space, does not provide strong delivery guarantees when the network is reconfigured with servers joining and leaving. This best-effort behavior is sufficient for eventually consistent data services such as key-value stores, content distribution networks or publish/subscribe systems. However, such schemes are not able to provide stronger consistency guarantees as required by, for example, metadata services and databases.**

We present RECODE, a framework for reconfigurable, consistent and decentralized data services. RECODE simplifies the implementation of strongly consistent data services, and continues to provide strong guarantees even during reconfiguration. More specifically, we introduce the *routecast* primitive which delivers messages for a key in the same total order, independent of the servers responsible for the key. We demonstrate the expressiveness and practical usability of RECODE by presenting three applications: a map of atomic registers, a set of distributed counters, and a lease management system. We evaluate the performance and elasticity of RECODE executing in a cluster.

I. INTRODUCTION

Distributed storage systems such as file systems [1], [2], key/value stores [3], [4] and partition and lease management services [5] all use a coordination service to keep track of metadata such as file locations, servers or data partitions. The coordination service is the authoritative source for performing updates to metadata. It avoids data inconsistencies by making all accesses atomic. For fault-tolerance and scalability, distributed coordination services replicate their state over a set of servers. To guarantee consistent or linearizable[1] access they coordinate all operations among the servers to make sure that they are executed in a total order. These type of guarantees can be implemented with replicated state machines (RSM) [6] or group communication systems (GCS) [7].

Partitioned data services provide storage and access to data associated with an identifier from a large name space. The identifiers and data are mapped to the servers contributing resources to the service. Each server or set of servers are responsible for a partition of the name space. Distributed Hash Tables (DHTs) is one way to implement a partitioned data service. A DHT is fully decentralized by using algorithms for adding and removing servers that only affect a small set of servers. Thus, while reconfiguration in a coordination service is global (affects all members), a reconfiguration in a DHT or partitioned service is local. This is the fundamental property that makes partitioned services scalable, all operations are local.

In order to access data, partitioned data services use key-based routing [8], [9]. An execution of *route*(key,msg), forwards a message msg towards some server that is responsible for the partition covering key in the name space. This routing method in combination with the DHTs partitioning and server management has been used to build, for example, key-value stores, publish/subscribe systems and distributed file systems. However, the semantics of *route* is often best-effort, meaning that two different servers can deliver a message for the same key. This occurs when the servers think that they are responsible for partitions that overlap in the name space. Overlapping responsibilities can happen after, for example, a perceived failure, based on inaccurate failure detectors, leading to reconfiguration [10], [11], [12]. Thus, a partitioned data service based on DHTs cannot provide strong consistency without modifications.

We propose RECODE[2], a fully decentralized system with a routing primitive that provides linearizable consistency even during run-time reconfiguration, i.e. when adding/removing servers or re-partitioning the name space. To achieve this, we combine the ability of RSMs to execute operations in a total order with consistent management of a global name space.

Our model of a partitioned data service differs from DHTs in that the name space partitioning does not change when servers join, leave or fail. Instead they join or leave a dynamic process group that implements an RSM[3]. We provide three operations for name space management: *split* (divide a partition), *merge* (combine two partitions) and *handover* (move a partition between servers). By making the name space management explicit, the group membership is separated from partition management. Thus, we can rely on well-known methods for dynamic process groups, e.g. [13] and [14].

To ensure consistent access to data items we introduce the key-based routing primitive *routecast*. Informally, the semantics of *routecast* guarantees that routed messages are delivered in a total order at each distinct name space key. This is valid even during split, merge and handover operations. We demonstrate how to use *routecast* to implement three high-level services: a map of atomic registers, distributed counters and a scalable lease management service [5].

In the remaining sections of this paper we present the

[1]Intuitively, linearizability means that a read always returns the last value written.

[2]Reconfigurable, Consistent and Decentralized

[3]Each server represents a process

related work before we discuss the system model and present the semantics of the partition management operations and the *routecast* primitive. This is followed by an implementation, with algorithms for the *routecast* primitive and the handover operation. In particular, we describe how the handover, split and merge can guarantee total order delivery of routed messages during reconfiguration. Finally, we evaluate the system in a proof-of-concept implementation in a cluster environment and demonstrate the implementation of three services on top of *routecast*.

II. Related Work

For DHTs, Shafaat et al. [10] showed that incorrect failure detectors may result in nodes believing that they are responsible for overlapping partitions. This conflict is eventually resolved by the maintenance algorithm, but even the shortest period of overlapping responsibilities may lead to data inconsistencies. A DHT does not correctly provide atomic data access, where clients must read the latest write, without significant modifications to the join and leave algorithms or the system model.

In order to provide data access with stricter consistency, the system must guarantee that each partition has an exclusive owner. That is, when a node delivers a routed message to the application, it must be the exclusive owner of the range containing the key. It was shown by both Ghodsi [12] and Risson [11], using the CAP theorem [15], that it is impossible to perform atomic changes to a ring topology without blocking while providing data consistency.

Lynch et al. [16] were first to suggest a solution to this problem by introducing a Replicated State Machine (RSM) as a fault-tolerant process (DHT node). With non-failing processes, there is no need for failure detectors and it is assumed that a process eventually responds to a request.

Both Ghodsi and Lynch et al. [12], [16] introduce protocols which assume that nodes are fault-tolerant. Lynch et al. propose an algorithm for join and leave similar to the approach in Chord [8]. Ghodsi's algorithm provides atomic change of a double-linked ring topology with fault-tolerant nodes. If nodes may fail, the algorithm becomes eventually consistent through a stabilization mechanism. The algorithm uses locks to indicate when a node is taking part in a topology change. Similar to Risson's approach, during a change, the successor node cannot deliver any requests to the application for the partition between the joining/leaving node's id and the successor's id.

Scatter [17] is a novel approach inspired by Lynch's RSM-based solution. This system has overlapping goals with Recode, it provides clients with linearizable consistency per key and it scales with the number of groups/servers while allowing run-time reconfiguration. RSMs are created dynamically by splitting an RSM into two or merging two consecutive RSMs into one while maintaining the successor and predecessor pointers for the ring structure. This is an interesting approach, however, merging and splitting RSMs leads to complex corner cases where RSMs must be initialized and shut-down on-the-fly as part of name space changes. Additionall, Scatter still suffers from the disadvantages of maintaining a ring structure. Modifications to the ring are done using a modified two-phase commit (2PC) requiring coordination between three RMSs.

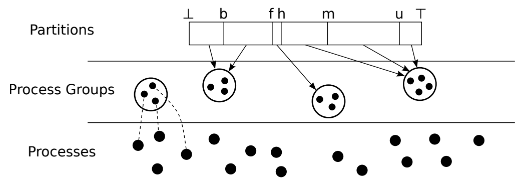


Fig. 1. System model overview with partitions, process groups and processes.

Unlike the previous approaches we do not maintain a ring. This simplifies the reconfiguration of the name space since there are no successor or predecessor pointers that must be updated atomically between groups. Also, node membership changes do not affect the partitioning of the name space. Instead, responsibility is decided based on an explicit range-to-server assignment. To change the assignment we introduce the *handover* algorithm, a bilateral agreement between two nodes in the system. Additionally, none of these approaches introduces well-defined semantics for delivery of a routed message.

The separation of the name space management and routing is also part of the design of key-value stores such as PNUTS [4] and BigTable [3]. However, consistency of name space operations are guaranteed through a single coordination service. Our algorithms enable this service to be partitioned at run-time.

III. RECODE Overview

A. System Model

The basic elements of the Recode model is a *partitioned name space* maintained by a set of *process groups* (Fig. 1). A partition is a range $[a, b)$ of identifiers in the name space. Each group mimics a single process by executing algorithms for a reconfigurable replicated state machine (RSM) [6], [14]. When a new process joins a Recode system it joins a process group using the mechanisms provided by the group. Similarily, when a process leaves or fails it is handled by the group.

A process group is exclusively *responsible* for a partition (or key/identifier) if the RSM maintains the state of the partition, i.e. both the metadata describing the partition and the data itself. Process groups may be responsible for zero or more partitions, but all partitions must be *assigned* to exactly one group. Thus, there are no two groups responsible for overlapping partitions and all partitions are assigned to some group (no gaps). We represent a partition as a $((a, b],k,G)$-tuple, where $(a, b]$ is the range of keys, k a version incremented on partition change and G the responsible (or owner) of the partition. The initial system state is a single partition covering the entire name space assigned to some process group, $((\perp, \top],1,G)$.

A significant difference in Recode compared to a DHT is that process membership is decoupled from the management of the name space partitions. In a DHT, each individual process is responsible for a partition. If a process becomes slow or the network drops messages, another process automatically takes over this process' partition through the DHT maintenance protocol. However, detecting a failure using a failure detector or through a periodic monitoring message may return an incorrect answer, e.g. a process may still answer requests from other processes without receiving the monitor request [10]. In the DHT model, this error may lead to the re-assignment of the responsibility for a partition from a still correct process without it even knowing that it is not responsible anymore. Thus, the system ends up in an incorrect state where two processes are responsible for an overlapping partition. Decoupling the responsibility revocation and assignment of partitions from the membership decision makes it possible to avoid this inconsistency.

Furthermore, this decoupling has three additional advantages. First, process groups may be responsible for more than one partition. This makes it possible to balance the load more fairly between groups as shown in [18], [8]. A fair load is necessary to efficiently use the system resources. Second, each group can have a different number of member processes. This can for example be useful if some partitions reside on more unreliable servers where higher reliability is required. Finally, when partitions are associated with state, the data movement between groups becomes an explicit decision instead of occurring each time a process fails, joins or leaves the system.

B. Modules and Interfaces

The main goal of RECODE is to provide three properties: reconfiguration, consistency and full decentralization. For this purpose we use four different modules: the process group, a routing service, partition management and *routecast* (Fig. 2). By clearly separating the modules with well-defined interfaces we can compose a system with different properties. For example, RECODE contains a module for routing messages towards a key in the partitioned name space. The cost of routing a message can be implemented to take, for example, $O(1)$ or $O(\log N)$-hops depending on network requirements.

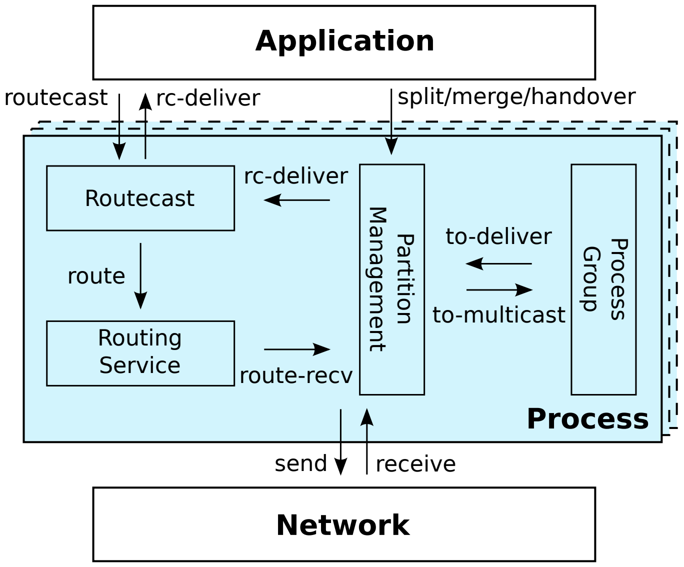


Fig. 2. Overview of the RECODE architecture.

An application implemented on top of RECODE uses *routecast* to send messages towards a key in the name space. Each event has a corresponding handler which is executed when the event is delivered at some process group responsible for the key. This allows us to implement both read and write operations unique to each key. Section 6 introduces three different applications built on top of RECODE.

Since *routecast* and the partition management module are the novel parts of RECODE, we introduce their interfaces and properties with further detail in this section. Assume for now that the process group exports an operation $to\text{-}multicast(m)$ which, informally, guarantees that all processes in a process group eventually delivers m and that two messages are delivered in the same total order to all processes. A completed $to\text{-}multicast$ triggers a $to\text{-}deliver$-event. The routing service provides a $route(x,m)$-method which forwards a message m towards the process group responsible for a partition covering the key x. When $route$ is invoked it eventually results in the execution of $route\text{-}receive$ at a process in the responsible process group. Unlike *routecast*, the routing functionality only requires the routing topology to converge eventually.

a) Routecast: The routing service enables key-based routing: given a key, a message is forwarded towards a process responsible for a partition covering the key. In a system with replicas where several processes are responsible for the same partition, a request requires coordination to provide consistent access. We introduce the *routecast* primitive which is based on the *route*-abstraction, but provides stronger guarantees. This module exports a *routecast* and the corresponding *rc-deliver*-primitives. $routecast(x,m)$ forwards a message m towards the process group responsible for x. Executing $routecast$ results in $rc\text{-}deliver(x, m, p)$ being invoked with a message m at each member of a process group A responsible for the partition covering the key x.

Intuitively, the *routecast* primitive ensures that messages for a given key x are delivered in a total order at the process group currently responsible for the partition covering x. With a total order of messages for a single key, an application can apply messages atomically to the value (or state) associated with this key. Furthermore, and unlike $to\text{-}multicast$, the $routecast$ primitive must ensure that messages are delivered in a total order even during changes of the partitioned name space. We call this property *partitioned total order delivery*. Below we specify all properties for $rc\text{-}deliver$. Note that p^k is partition p at version k.

- *PTO1 Partitioned Validity* If a *correct* process executes $routecast(x, m)$, then some process group A eventually executes $rc\text{-}deliver(x,m,p)$ for a partition $([a, b), k, A)$ such that $x \in [a, b)$.
- *PTO2 Partitioned Integrity* A routed message (x,m) is only rc-delivered if some process executed $routecast(x, m)$ and it is rc-delivered at most once.
- *PTO3 Same Partition Total Order* For any pair of routed messages (x,m) and (y,m') that are rc-delivered in a partition p, there exist a total order $\prec_p$ such that if $(x,m) \prec_p (y,m')$, then $rc\text{-}deliver(x,m,p)$ is executed before $rc\text{-}deliver(y,m',p)$.
- *PTO4 Last Partition Delivery* If a process group invokes $rc\text{-}deliver(x,m,p^k)$, then there exists no partition $q^{k'}$, where $x \in q^{k'}$ and $k' > k$.
- *PTO5 Partitioned Total Order* For any pair of routed messages (x,m) and (x,m'), there exist a total order $\prec_x$ such that if $(x,m) \prec_x$

(x,m') then $rc\text{-}deliver(x,m,p^k)$ is executed before $rc\text{-}deliver(x,m',q^{k'})$, where x is covered by p, q and $k \leq k'$.

These properties define that there is a total order on the partitions that cover x. That is, if p^k and q^{k+1} both cover x, than any routed message for x is delivered to the owner of the partition according to a total order. Furthermore, if two routed messages for x and y are delivered within the same partition, then they are also delivered in a total order. Because of *PTO3*, the delivery to elements in the same partition becomes dependent. A weaker specification that replaces this property with one that makes delivery to different elements independent would allow for more parallelism.

b) Partition Management: The partition management module maintains the state of local partitions that a process is responsible for. It provides three operations on this state: $split(x, p)$, $merge(p, q)$ and $handover(x, k)$. A split divides a partition into two non-overlapping partitions and a merge combines two adjacent partitions into one. The handover changes the ownership from one process group to another, which makes it possible to increase or decrease the set of groups responsible for partitions at system run-time.

The partition management operations must guarantee that two different groups are never responsible for the same partition or an overlapping partition. Furthermore, for partitioned total order, any element in the name space must be covered by a partition (no gaps). We define the properties of the partition management operations as follows:

- *P1 Exclusive Assignment* No two process groups are responsible for the same element x in the name space.
- *P2 Handover Validity* For any handover, it is initialized by some process and it terminates at most once.
- *P3 Handover Termination* A handover of a partition p between two process groups A, B, eventually terminates with either A or B responsible for p.

Property *P1* defines that a successful assignment change for a partition cannot lead to two different groups being responsible for an overlapping name space range. The change itself must be atomic, but the new owner only needs to be responsible for the partition p at the time of termination, *P2*. *P3* avoids that a handover is decided on twice, effectively, re-assigning a partition to the same owner. Since *P3* allows either A or B to be responsible after termination, this property enables the current owner to reject a handover request.

IV. Implementation

We describe the distributed system model followed by the algorithms necessary to implement both *routecast* and the management operations: split, merge and handover of partitions.

A. Distributed System Model

Processes execute in a partially synchronous model such as the timed-asynchronous model [19], where it is possible to solve the consensus problem (and consequently TO-multicast). A message is sent by a process using *send*(m) and received with *receive*(m). Communication between processes is quasi-reliable [20] and we expect that unless the communicating processes are faulty, a message eventually arrives, and is received at most once, even though the underlying channel may lose, delay or re-order messages.

B. Reliable Process Groups

There are essentially two main techniques used to implement reliable process groups with gap-free total order broadcast (RSM): destination agreement and primary sequencer [21]. In destination agreement, a set of processes uses consensus to agree on the message in an instance and then on the order of the instances. Any process in the group can propose a message in an instance and if a majority agrees, the message can be delivered. A popular approach to implement destination agreement is Paxos [22].

Primary sequencer enforces the message order through a designated primary process. To reliably store the operations, the primary writes to at least a majority of backups. However, this approach requires additional protocols for primary failover and catch-up of backups that have fallen behind. A recent primary/backup protocol is Zab [23] which is used in the ZooKeeper coordination service [24].

Both approaches provide the same semantics, but with different failure and performance characteristics. With destination agreement, any process can initiate an operation while with a primary sequencer, the primary must be available. However, the benefit of the primary approach is that in the failure-free case, a write only requires a single round-trip plus a non-blocking notification, and a read can be replied to without communication with the backups. In destination agreement, both read and writes require two round-trips in the non-optimized case. By introducing master leases as in Chubby [25], the two approaches become identical. Our proof-of-concept implementation is based on primary/backup, since *routecast* requests require consistent access to the partition state. Dynamic group membership for reconfiguration of the group is achievable through complementary protocols such as SMART [14] or [13].

In addition to adding and removing processes to an existing group, we assume that there is functionality to initialize new groups and disband existing groups. This is necessary to scale up or down the system. After a group has disbanded it does not reply to any more requests. Disbanding a group is only allowed if the group is not responsible for any partitions, since this would otherwise lead to a gap in the partition name space.

C. Routing Service

The routing service exports the *route*(k, m) primitive which is used to locate the process group currently responsible for a partition covering a given key. That is, by forwarding a message towards the key it will eventually be received by a process part of the responsible process group. An advantage from this abstraction is that the routing implementation does not need to know about process groups. Messages forwarded using the *route* function are sent directly between processes based on the topology.

When the system is reconfigured by moving partitions between process groups, the topology must be adapted accordingly. Unlike the partition management, the topology can be updated to eventually reflect the most current partition assignment. Each process part of a process group has access to the current partition state of that group and uses this to update the routing service. The routing function can be implemented as a library at the application clients, as a separate service with dedicated servers or as part of the processes in the process group and the topology can be of any type, e.g. ring (DHT), complete map (O(1) routing) or a tree (DNS), as long as a routed message eventually arrives at the responsible group.

D. Routecast

To implement the *routecast* primitive, we use the functions exported by the routing service, *route*, and the process group, *to-multicast*. Based on these abstractions and the partition state maintained by each process group, the algorithms presented in Alg. 1 are straight-forward. We assumed that the system is initialized with a partition covering the entire name space at some process group.

Algorithm 1: The algorithm used to execute *routecast*.

```
1 procedure routecast($x$,$msg$) do
2     route($x$, RCREQ($x$,$msg$))
3 on route-receive RCREQ($x$,$msg$) for $x$
4     to-multicast(RCREQ($x$,$msg$))
5 on to-deliver RCREQ($x$,$msg$) do
      ▷ Are we responsible for the partition covering $x$?
6     if $x \in p \in partitions$ then
7         rc-deliver($x$,$msg$,$p$)
```

To execute a *routecast* operation, we route a RCREQ(k,msg) message towards the owner by using the routing service. When *route-receive* executes for this message a *to-multicast* is invoked with the request (line 4). On the execution of *to-deliver*, the *partitions* variable is used to check if the process group is still responsible for the partition, and in that case, *rc-deliver* executes with the wrapped message, the key and the partition the message is delivered in.

E. Partition Management

Managing the partition name space is done using three operations: *split*, *merge* and *handover*. With these operations, the system can grow and shrink both the name space as well as the set of process groups responsible for partitions. In order to provide the guarantees of *routecast* all name space changes must be perceived atomic. Moreover, partitioned total order delivery requires that operations performed to an element in the name space are totally ordered, independent from which process group is responsible for the covering partition. To this end, the main idea behind the partition management operations is to ensure causality of any changes. Practically this is done by associating each partition with a version that is always increasing and inherited from parent partitions, i.e. partitions with an overlapping range and lower version number. When a group is responsible for a partition it has the exclusive right to increase its version and modify the partition's range and owner. Atomicity is achieved by using the groups *to-multicast*-primitive, which interleaves any partition management operations with *routecast* requests.

1) Split and Merge: Initially, the system starts with a single group storing a partition covering the entire name space in a *partitions* variable and partitions are represented with ($[a,b)$,$version$,$group$)-tuples. A split-operation divides a partition into two new partitions with consecutive ranges and increased versions. Similarly, a merge combines two partitions with consecutive ranges into a single partition and a version which is larger than both of the parents. Both operations assumes that the partitions involved are stored in the local *partitions* variable at the same process group. We thereby avoid any coordination across process groups when splitting and merging partitions.

After a successful split or merge, the parent partitions are removed from the *partition* variable while the new partitions are added. Since both operations are atomic (with *to-multicast*, several updates to the local state can be performed in a single *to-deliver* invocation) and applied to consecutive ranges at the same process group, they cannot produce a name space with gaps or overlaps.

2) Handover Algorithm: A process group responsible for a partition has exclusive access to perform any modifications to that partition. A handover transfers the exclusive access between process groups. The idea behind the handover algorithm (fig. 3 and alg. 2) is to let the current owner update the partition in its local state with the new owner and then telling the new owner that it is responsible. Intuitively, the new owner *steals* the partition from the current owner. Since the owner change operation is executed within the process group, the change is atomic.

There are two complications to the handover protocol. First, we allow process groups to disband at any time as long as they do not store any state. Second, we must handle concurrent handover requests. The protocol has two phases: the handover and the clean-up phase. The clean-up makes sure that all state at the group responsible for the partition before the handover is removed. Without local state, the group can disband and leave the system. We describe the handover algorithm in detail below.

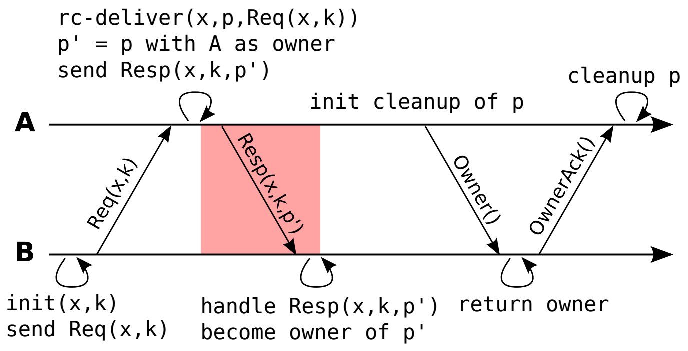


Fig. 3. Hand-over of a partition p to A.

The group maintains two sets for ongoing handovers: the *active* set contains (x,k)-tuples, where x is a key and k is a proposed partition version number for an ongoing handover.

The role of $active$ is to avoid that the same handover is concurrently initialized and terminates more than once (property *P2*) and to protect against group disbands. $pending$ is a set of partition-tuples with all partitions that can be garbage collected in the clean-up phase.

In the protocol, a process group A tries to retain ownership of a partition p owned by B, where $x \in p$. The handover phase is started from A and the clean-up phase is initiated from the responsible of x after a successful handover. The handover phase ensures that the current owner of p, B, gives up the responsibility of p and that A becomes the new owner.

The handover request contains $x \in p$ and a proposed new version k for p. Similar to an acceptor in Paxos [26], a process group delivering a valid handover request must always accept the request if k is higher than the current version of p, $p_{version}$ (line 10). An accepted handover results in an atomic change of owner when executing rc-$deliver$ (line 11-14). After this change, the group will not deliver any more messages within the partition. Thus, for concurrent handover requests the first request delivered and accepted "wins" the partition. Note that rc-$deliver$ will not execute for p again after it has been removed from $partitions$. This atomic change of ownership guarantees the exclusive assignment property *P1* and that either A or B is responsible for p after termination *P3*.

In the clean-up phase, B tries to ensure that the handover of p has terminated. That is, the handover phase has completed when a new process group stores p in the $partitions$ table, and is thereby able to rc-$deliver$ messages for p. This may be another group than A, since A could have received a new handover request in the mean time. When B has completed the clean-up it is free to disband unless it is responsible for other partitions or is part of another handover operation indicated by the $active$ or $pending$ variables.

a) Avoiding Process Group Disband: Most of the complexity of the protocol is due to the possibility that a process group disbands concurrently with a handover. First, assume a simple RPC-based protocol where B tries to handover a partition to A by sending a HANDOVERREQUEST directly to A and waiting for a HANDOVERREPLY. Once the HANDOVERINIT that was invoked to send the HANDOVERREQUEST has finished executing at B, a HANDOVERREPLY must eventually be received by B in order to terminate the protocol at B. However, if A decides to disband before it has received the HANDOVERREQUEST from B, then B will wait indefinitely for a HANDOVERREPLY. B can not try to handover the partition to another process group, C, after a time-out since A could be temporarily unavailable or the routed message or the reply can be delayed. Thus, B cannot deterministically terminate the protocol. Initiating the handover from A to a fixed process group B suffers from the same problem.

To solve this issue, we 1) introduce the $active$-set and the clean-up phase to avoid that a group disbands while being responsible for system state and 2) use a $routecast$ request to initialize the handover. With $routecast$, the execution of rc-$deliver$ always occurs at the group currently responsible for p, which must be in the system since it has state. This effectively avoids that B will wait indefinitely for a HANDOVERREPLY.

Algorithm 2: Handover algorithm for a partition p

```
1  $active \leftarrow \emptyset$            ▷ Active (not terminated) handovers.
2  $pending \leftarrow \emptyset$           ▷ Pending handover verification
3  procedure handover(x,k) do
4      to-multicast(HANDOVERINIT(x,k))   ▷ Handover with key x and version k.
5  on to-deliver HANDOVERINIT(x,k) do
6      if $(x,k) \notin active$ then
7          $active \leftarrow active \cup \{(x,k)\}$
8          routecast HANDOVERREQUEST(x,k,self) towards $x$
9  on rc-deliver HANDOVERREQUEST(x,k,group) for $x \in p$ do
10     if $k > p_{version}$ then   ▷ Always accept a higher proposed version
11         $p' \leftarrow (p_{start}, p_{end}, k, group)$
12         $pending \leftarrow pending \cup \{p'\}$
13         $delete\ partitions[p_{range}]$
14         send HANDOVERREPLY(x,k,p') to group
15     else            ▷ $k$ is less than the latest partition version
16         send HANDOVERREPLY(x,k,p) to group
17 on receive HANDOVERREPLY(x,k,p) do
18     to-multicast(HANDOVERREPLY(x,k,p))
19 on to-deliver HANDOVERREPLY(x,k,p) do
20     if $(x,k) \in active$ and $self_{id} = p_{group_{id}}$ and $p_{version} = k$ then
21         $partitions[p_{range}] \leftarrow p$
22     $active \leftarrow active \setminus \{(x,k)\}$   ▷ Handover has finished.
```

b) Partition Availability: The presented algorithm trades availability for consistency. Messages for $x \in p$ can continue to be rc-delivered at an owner B until the HANDOVERREQUEST has been delivered. At this point, the ownership of the partition is either changed to A or stays with B depending on the value of the proposed version k. For a successful handover, in the time between the delivery of the HANDOVERREQUEST at B and the *to-deliver* of the HANDOVERREPLY message at A, no group can deliver messages for the partition. This means that any message is delayed until the HANDOVERREPLY has been to-delivered. In the failure-free case, this time is the latency of a single message send and a to-$multicast$ execution at A.

V. EVALUATION

The handover algorithm is the only operation that requires coordination outside the process group. We analyze the different costs and the fault-tolerance of a handover compared with three other approaches, Risson's [11], Ghodsi's [12] and Scatter [17]. We follow that with two proof-of-concept experiments that evaluates 1) the scalability of the *routecast* primitive and 2) the run-time reconfiguration of the name space.

A. Handover Costs

Table I compares the handover protocol with Risson's protocol called FTAR which uses Fast Paxos Commit, Ghodsi's optimized version of atomic ring maintenance and Scatter. We look at five different costs: 1) the total number of messages, 2) the number of message delays, 3) the number of message delays for which a partition is unavailable for the delivery

	1) Messages	2) Delays	3) Delays unavailable	4) Processes	5) Process failures
RECODE	$2r_m + 2 + 3g_m$	$2r_d + 2 + 2g_d$	$1 + g_d$	2 groups	$2F + 1$
FTAR	10	4	4	3 processes	1
Atomic join	$5(1 + g_m)$	$5(1 + g_d)$	$4(1 + g_d)$	3 groups	$2F + 1$
Atomic leave	$6(1 + g_m)$	$6(1 + g_d)$	$6(1 + g_d)$	3 groups	$2F + 1$
Scatter	$3g_m + 2g_m$	$4 + 3g_d + 2g_d$	$2 + 2 + g_d$	3 groups	$2F + 1$

TABLE I. COST COMPARISON BETWEEN THE HANDOVER PROTOCOL, FTAR, ATOMIC RING (JOIN/LEAVE) MAINTENANCE AND SCATTER.

of routed messages, 4) the number of processes (or process groups) involved in a name space reconfiguration and, finally, 5) max processes that can fail. RECODE, Atomic Ring and Scatter require fault-tolerant processes (RMSs) for correctness in case of failure, we denote the cost of an operation in an RSM by g_m and the message delays by g_d.

Both phases in the handover protocol include a routing step. The cost of routing varies depending on which routing algorithm and topology is used, we denote the number of messages as r_m and r_d as the message delays. The total number of messages for the handover phase and the clean-up phase is $2r_m+2$, one routing step and the corresponding reply. Similarly, the message delays are $2r_d + 2$. During a handover, requests to a partition are delayed with one message delay (the reply to the handover request) and one *to-multicast*, before the new owner is responsible and can start answering requests. Finally, there are only two participants involved in the protocol if we ignore the processes executing the route-requests.

Atomic Ring, RECODE and Scatter depend on fault-tolerant process groups, which incurs significantly more messages and message delays when compared to FTAR. However, FTAR only allows one failure. Atomic Ring, Scatter and RECODE can handle F failures in a group with $2F + 1$ processes. We also note that in an efficient state machine implementation, for example with primary/backup, only 2 message delays are necessary to execute an operation [27]. Thus, with an efficient implementation a handover in RECODE only delay data operations for 3 message delays.

B. Implementation and Experiment Setup

We have implemented a proof-of-concept of RECODE using Scala. Each process group member runs in a single JVM and receives messages from the network layer (NIO + Netty) with a single thread. A process group exports a *to-multicast* primitive using a primary/backup-based implementation. New members can be added and existing members removed using the group membership protocol. Primary fail-over uses a lease mechanism as described in [28]. All state is in memory, which makes the system correct as long as there are no power failures affecting a majority of processes in a group[4]. The primary executes one *to-multicast* request at a time and new requests are placed in a FIFO queue. The rate of *to-multicast* requests is bounded by the network latency, a majority of group members must ack each multicast request from the primary before the next is executed. Thus, an operation is stable after a single round-trip (2 message delays). We note that there are several techniques to improve the performance of a single process group, such as pipelining and batching of messages [29], [27]. However, we keep the group simple to make the analysis of the systems performance easier.

[4]There were no power outages during the experiments.

For each of the experiments presented below, we have clients issuing requests synchronously. We measure the throughput of each client as the completed number of requests per second. The latency is the time from the start of the request until it returns (i.e. the time the client blocks).

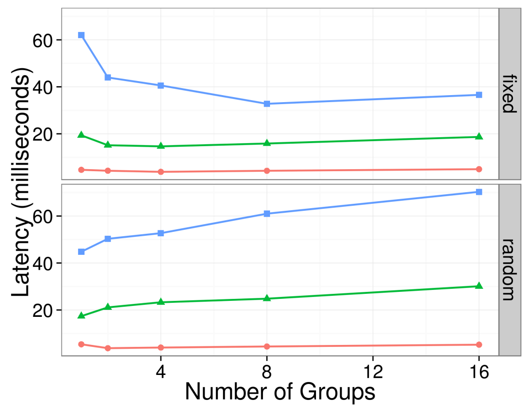
We have implemented a simple routing service where each router has a full mapping from partition to process group. On any change: group view change, partition handover, split or merge, the primary in the process group sends an update message to the routing service which broadcasts the update internally to all routers. Although this approach is not scalable to 1000s of servers, it works well for smaller systems. Clients are routing messages iteratively, that is they send a request for a key and wait for a reply with the process group responsible for the partition covering the key. Requests to a non-primary are forwarded to the primary without contacting the client. When using the router a client request needs three message delays before reaching the process group responsible for a key. To avoid re-doing look-ups for a known key or partition, the client keeps a cache of partitions. An entry is invalidated when the process group no longer is responsible for the partition due to a handover.

The experiments are executed on a cluster with 32 machines connected with a 1Gbps Ethernet switch. The average latency between any two machines is 0.15 ms. We use the 64-bit OpenJDK JVM build 14.0-b16 and Scala 2.9.0-1.

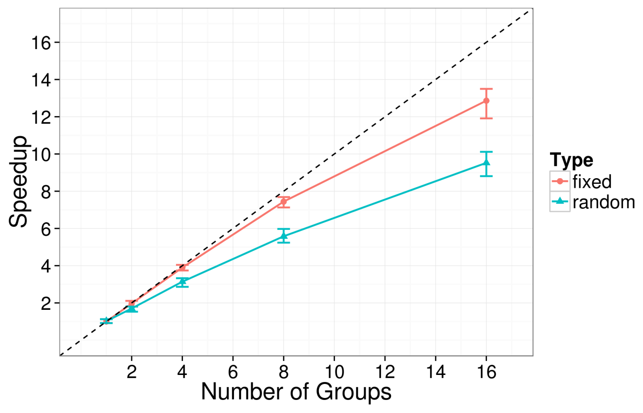
C. Scalability

With a partitioned name space as in RECODE, it is possible to spread the load evenly across both partitions and process groups and thereby servers. With even load over the groups and partition boundaries corresponding to the workload pattern, the system should scale linearly. We evaluate this claim by comparing our implementation throughput with the expected speedup. The speedup is calculated as $speedup(n) = \frac{tput(n)}{tput(1)}$, where n is the number of groups and $tput(n)$ is the average throughput for n groups. A single group handles around 1000 operations/second with the current implementation.

We have two different client workloads, fixed or randomly generated keys. In the fixed strategy, one or more clients send *routecast* requests for the same key as fast as possible. Thus, each request always ends up at the same group responsible for the key. At the group a TO-request is generated for each request and it is sufficient with five (because of round-trip time) clients to always have one in-flight request within the group (pending requests are queued). Thus, with two groups we need 10 clients and so on. Using a random workload we emulate equally balanced partitions, however, all clients send requests to all process groups unlike in the fixed case.



(a) Latency



(b) Throughput

Fig. 4. Throughput and latency with an increasing number of groups and clients for *routecast* and multicast.

The clients, groups and routers are all executing on different machines. Members of the same group are always on different machines, but the same machine may have several processes. 21 machines are used for hosting group member processes, 5 for clients and 2 for routers. We measure the latency (50th, 95th and 99th percentile) and throughput of requests at the clients.

Figure 4 shows the latency and speedup for an increasing number of groups with the different workload strategies. We observe that for the fixed cases, the latency for all percentiles is stable with an increasing number of groups and the throughput increases linearly initially. The overhead in both the latency and speedup comes from the increased number of clients necessary to saturate the queue. The decrease in speedup for 8 and 16 groups is attributed to that multiple processes are using the same physical machine. Each process uses the same network card and sends many small packets. Additionally, the networking library, netty, has a thread pool for managing connections. With two or more processes per machine we observe a super-linear increase in context switches, which is likely to explain the performance reduction. Since the throughput is latency-bound, variations when accessing the underlying hardware have a larger effect on the overall throughput.

With random requests there is a higher chance that the queues at the primaries are unbalanced. This causes two things, first, the variation of latency increases since requests may stay in a long queue. Second, the probability of an empty queue increases which leads to a sub-linear speedup. An increasing number clients also leads to more variation and does not guarantee that all queues are busy. A solution to this problem is to introduce pipelining, where several requests are executed concurrently.

D. Elasticity

The mechanisms for partition management enables the application to allocate and de-allocate resources at run-time by splitting, merging and moving partitions between groups. We evaluate this mechanism in a single experiment by measuring the throughput and latency over time. The system runs with a fixed number of clients (74), groups (4) and partitions (16), after 180 seconds we add 4 more groups and start balancing. A balancing operation uses the handover mechanism to transfer a partition from the group with the most partitions to the group with the least number of partitions. We execute 8 balance operations with a 60 second interval between each operation.

Figure 5 contains two plots with the latency and throughput of the system during 15 minutes. From the latency graph in fig. 5a, we can see that the system is overloaded initially. With four groups and 74 clients, the queue of TO-multicast operations at each group has a high variance due to the random client requests. After the balancing has finished (around 600s), the requests are more evenly distributed and have more queues to choose from which results in reduced latency, variance and higher throughput (c.f. fig. 5b). The sudden drop in throughput and increased variance in latency when balancing starts is an effect from the redistribution of requests. Before balancing, each group has four equally sized partitions. The first balance operations places one partition at each new group, this leads to a higher unbalance in the group queues and temporarily increased variation and reduced throughput. However, after all 8 balance operations (around 600s) the system quickly stabilizes.

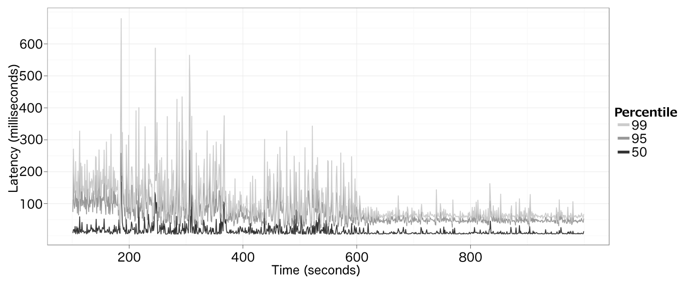
VI. Using RECODE

In this section we introduce three example applications and their implementation on top of the *routecast* primitive. The first application is a map with atomic registers, the second a distributed counter service and the third is a lease management service.

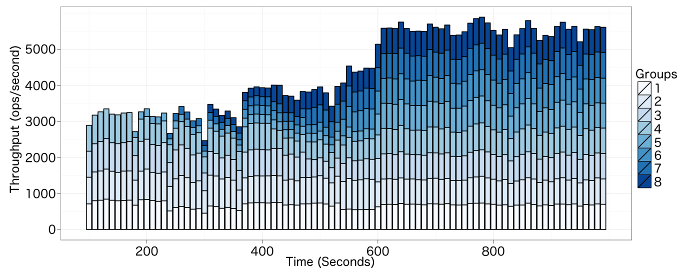
Since *routecast* is based on total order broadcast, it has the same guarantees as ZooKeeper [24] or Chubby [25]. It is therefore possible to implement the more general coordination service interface they are providing. However, unlike when using ZooKeeper and Chubby, RECODE is used as a library with the application code implementing handlers for $rc\text{-}deliver$ events and using *routecast* to send messages with strong guarantees.

A. A Map of Atomic Registers

In this section we describe how to implement a map of atomic read/write objects or registers. A map of atomic



(a) Latency



(b) Throughput

Fig. 5. Latency and throughput when partitions are re-allocated to new process groups.

registers is the basic data structure for building a distributed key/value-store. Each register is referenced using a key and is accessed atomically using the *routecast* primitive. A single atomic register has the following semantics [30]. *Termination*, every operation eventually completes. *Validity*, every read returns the last value written and *Ordering*, if a read returns v_2 after a read that precedes it has returned v_1, then v_1 cannot be written after v_2.

Algorithm 3: Read/write register implemented with *routecast*.

```
1 $registers \leftarrow \{\}$                ▷ Register id ↦ register value
2 on rc-deliver READ(x) for $x \in p$ do
3     return $registers[x]$ orElse $\perp$
4 on rc-deliver WRITE(x,v) for $x \in p$ do
5     $oldval \leftarrow registers[x]$ orElse $\perp$
6     $registers[x] \leftarrow v$
7     return $oldval$
```

A map with atomic registers is straight-forward to implement with a READ(x) and WRITE(x,v)-operation on top of *routecast*. Algorithm 3 presents the implementation of a read/write-register. The correctness depends on $rc\text{-}deliver$ which guarantees total order delivery for each element in the name space (*PTO5 Partitioned Total Order*). The efficiency of this implementation depends on the cost of achieving total order since each read/write is ordered through to-multicast. A primary/backup-based implementation can, for example, return a read directly from the primary and the performance is then depending on the read/write ratio. Partitioning of the register map does not have any dependencies at the data level since the resources are independent. An appropriate partitioning is likely to take into account the access frequency of the registers and the size of each value.

B. Distributed Counters

A distributed counter service is used to maintain statistics or aggregates for different resources. Such statistics is commonly used at, for example, web-based application that keep track of link clicks, played videos or songs. Each request increments an integer. Note that this cannot be implemented with a read/write register in a single operation (see section VI-A), since the increment first need to read the value to know what to increases, then adds one and finally overwrites the value. To handle this type of read-modify-write operations, we would need to extend a read/write register to somehow handle concurrency. However, with *routecast* the $rc\text{-}deliver$ execution is atomic for the delivered message and the increment can therefore execute without handling concurrency explicitly. It is already done through the ordering of operations by the TO-multicast implementation.

Algorithm 4: A service for distributed counters.

```
1 $counters \leftarrow \{\}$                ▷ Resource ↦ count
2 on rc-deliver INCREMENT(x) for $x \in p$ do
3     $count \leftarrow counters[x]$ orElse $0$
4     $counters[x] \leftarrow count + 1$
5     return $count$
6 on rc-deliver COUNTFOR(x) for $x \in p$ do
7     return $counters[x]$ orElse $0$
```

Algorithm 4 presents the implementation of a distributed counter service using the *routecast* primitive. The service increments the counter for a resource k when executing $rc\text{-}deliver$(INCREMENT(k)). The increment returns the old value of the counter to the client. To read the value of counter k, the client executes *routecast*(COUNTFOR(k)).

C. Lease Management Service

A lease is a time-based lock, that is, it grants exclusive access to some resource for a defined time. A lease management service such as Chubby [25], issues leases to clients that needs to access some resource in the system. A lease-request always returns the current valid lease, this lease can be held by the client making the access or another client. Internally, the service must first test if there exists a lease and return it, or issue a new lease. As the counter service, this also requires a read-modify-write-operation (test-and-set).

Algorithm 5 presents the lease management service implementation. To try to become the lease holder, a client executes *routecast* with GETLEASE(k,*client*), where k identifies the lease and $client$ the process trying to acquire the lease. Concurrent clients will be arbitrated depending on the total ordering of the TO-multicast used by $rc\text{-}deliver$. A client already holding a lease can renew it by trying to get the lease before the lease time expires.

VII. CONCLUSION

We have introduced RECODE, a system for total order delivery of messages in a run-time reconfigurable name space.

Algorithm 5: A Lease Management Service implemented with *routecast*.

```
1  $leases \leftarrow \{\}$                ▷ Lease id $\mapsto$ (timestamp, owner)
2  $t_{lease}$                             ▷ Time a lease is valid.
3  on rc-deliver GETLEASE($x$, $client$) for $x \in p$ do
4      $\lambda \leftarrow leases[x]$
5      if $\lambda = \bot$ or $\lambda_{timestamp} \geq now()$ then
           ▷ Invalid or non-existing lease, create a new lease.
6          $\lambda \leftarrow (now() + t_{lease}, client)$
7      else if $\lambda_{timestamp} < now()$ and $\lambda_{owner} = client$ then
           ▷ Lease renewal by the current owner.
8          $\lambda \leftarrow (now() + t_{lease}, client)$
9      $leases[x] \leftarrow \lambda$
10     return $\lambda$
```

We provide clear semantics of the *routecast* primitive and algorithms for reconfiguration and routing. The proof-of-concept implementation shows that the system is both scalable and can handle addition and removal of resources while still being available for requests. To exemplify the simplicity of the *routecast* abstraction, three applications were designed on top of RECODE with varying complexity. First, we implemented a map of atomic registers which can be used as a basis for a scalable and consistent key/value-store. Second, we presented a distributed counter service, which addresses a common problem for web-sites keeping aggregate statistics. Finally, we gave an implementation of a lease management service. Leases are often used to guarantee exclusive access to items in storage systems such as files or blocks. The lease service becomes a bottleneck when serving many small files [31], [28]. However, with RECODE it is possible to dynamically add more resources and perform the name space partitioning, which is necessary to scale the application at run-time.

ACKNOWLEDGMENTS

The authors would like to thank the reviewers for thorough and useful comments. We also thank Thorsten Schütt, Björn Kolbeck and Seif Haridi for fruitful discussions. Part of this work was supported by the EU projects Contrail, Harness and iSocial.

REFERENCES

[1] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *SOSP*. ACM, 2003, pp. 29–43.

[2] F. Hupfeld, T. Cortes, B. Kolbeck, J. Stender, E. Focht, M. Hess, J. Malo, J. Martí, and E. Cesario, "The XtreemFS architecture - a case for object-based file systems in Grids," *Concurrency and Computation: Practice and Experience*, vol. 20, no. 17, pp. 2049–2060, 2008.

[3] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Trans. Comput. Syst.*, vol. 26, no. 2, 2008.

[4] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni, "PNUTS: Yahoo!'s hosted data serving platform," *PVLDB*, vol. 1, no. 2, pp. 1277–1288, 2008.

[5] A. Adya, J. Dunagan, and A. Wolman, "Centrifuge: Integrated lease management and partitioning for cloud services," in *NSDI*. USENIX Association, 2010, pp. 1–16.

[6] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Comput. Surv.*, vol. 22, no. 4, pp. 299–319, 1990.

[7] K. P. Birman, "The process group approach to reliable distributed computing," *Commun. ACM*, vol. 36, no. 12, pp. 36–53, 1993.

[8] I. Stoica, R. Morris, D. R. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *SIGCOMM*, 2001, pp. 149–160.

[9] T. Schütt, F. Schintke, and A. Reinefeld, "Range queries on structured overlay networks," *Computer Communications*, vol. 31, no. 2, pp. 280–291, 2008.

[10] T. M. Shafaat, T. Schütt, M. Moser, S. Haridi, A. Ghodsi, and A. Reinefeld, "Key-based consistency and availability in structured overlay networks," in *HPDC*. ACM, 2008, pp. 235–236.

[11] J. Risson, "Reliable Key-Based Routing Topologies," Ph.D. dissertation, The University of New South Wales, 2007.

[12] A. Ghodsi, "Distributed k-ary System: Algorithms for distributed hash tables," PhD Dissertation, KTH—Royal Institute of Technology, Stockholm, Sweden, Oct. 2006.

[13] A. Schiper, "Dynamic group communication," *Distributed Computing*, vol. 18, no. 5, pp. 359–374, 2006.

[14] J. R. Lorch, A. Adya, W. J. Bolosky, R. Chaiken, J. R. Douceur, and J. Howell, "The SMART way to migrate replicated stateful services," in *EuroSys*. ACM, 2006, pp. 103–115.

[15] S. Gilbert and N. A. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *SIGACT News*, vol. 33, no. 2, pp. 51–59, 2002.

[16] N. A. Lynch, D. Malkhi, and D. Ratajczak, "Atomic data access in distributed hash tables," in *IPTPS*, vol. 2429, 2002, pp. 295–305.

[17] L. Glendenning, I. Beschastnikh, A. Krishnamurthy, and T. E. Anderson, "Scalable consistency in scatter," in *SOSP*. ACM, 2011, pp. 15–28.

[18] J. Ledlie and M. I. Seltzer, "Distributed, secure load balancing with skew, heterogeneity and churn," in *INFOCOM*. IEEE, 2005, pp. 1419–1430.

[19] F. Cristian and C. Fetzer, "The timed asynchronous distributed system model," *IEEE Trans. Parallel Distrib. Syst.*, vol. 10, no. 6, pp. 642–657, 1999.

[20] M. K. Aguilera, W. Chen, and S. Toueg, "Using the heartbeat failure detector for quiescent reliable communication and consensus in partitionable networks," *Theor. Comput. Sci.*, vol. 220, no. 1, pp. 3–30, 1999.

[21] X. Défago, A. Schiper, and P. Urbán, "Total order broadcast and multicast algorithms: Taxonomy and survey," *ACM Comput. Surv.*, vol. 36, no. 4, pp. 372–421, 2004.

[22] L. Lamport, "The part-time parliament," *ACM Trans. Comput. Syst.*, vol. 16, no. 2, pp. 133–169, 1998.

[23] F. Junqueira, B. Reed, and M. Serafini, "Zab: High-performance broadcast for primary-backup systems," in *IEEE DSN*, June 2011.

[24] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: wait-free coordination for internet-scale systems," in *Proceedings of the 2010 USENIX ATC Conference*. Berkeley, CA, USA: USENIX Association, 2010, pp. 11–11.

[25] M. Burrows, "The chubby lock service for loosely-coupled distributed systems," in *OSDI*. USENIX Association, 2006, pp. 335–350.

[26] L. Lamport, "Paxos made simple," *ACM SIGACT News*, vol. 32, no. 4, pp. 18–25, 2001.

[27] T. D. Chandra, R. Griesemer, and J. Redstone, "Paxos made live: an engineering perspective," in *PODC*. ACM, 2007, pp. 398–407.

[28] B. Kolbeck, M. Högqvist, J. Stender, and F. Hupfeld, "Flease - Lease Coordination without a Lock Server," in *25th IEEE International Parallel and Distributed Processing Symposium, IPDPS, 2011.*

[29] P. J. Marandi, M. Primi, and F. Pedone, "High performance state-machine replication," in *IEEE Int'l Conf. on Dependable Systems and Networks*, June 2011.

[30] R. Guerraoui and L. Rodrigues, *Introduction to reliable distributed programming*. Springer, 2006.

[31] M. K. McKusick and S. Quinlan, "Case study: GFS: Evolution on fast-forward," *ACM Queue: Tomorrow's Computing Today*, vol. 7, no. 7, p. 10, Aug. 2009.