# A Consensus-Based Reconfigurable Group Communication System

Hans P. Reiser[1], Udo Bartlang[2], and Franz J. Hauck[1]

[1] {reiser,hauck}@informatik.uni-ulm.de
Distributed Systems, University of Ulm
[2] udo@bartlang.de
Department of Computer Sciences, University of Erlangen-Nürnberg

**Abstract.** Group communication is an essential building block for the development of fault-tolerant distributed applications. This paper presents a reconfigurable totally-ordered group communication system based on distributed consensus algorithms. Our novel design uses a policy-based mechanism for dynamical reconfiguration of the system at runtime without service interruption. Reconfigurations may optimize for most efficient "best-case" operation or for minimal delays in failure situations, may select different failure models like crash-stop, crash-recovery, or Byzantine, and may adjust internal parameters like timeout values for failure detection. Internally, our modular group-communication system is composed of a low-level transport, the group management, and an instance of a distributed consensus algorithm. Performance measurements of our Java implementation illustrate the practical feasibility of our approach.

## 1 Introduction

Fault tolerance is an essential challenge in the development of distributed systems. The inherent complexity of the development of fault-tolerant systems demands the support by a middleware infrastructure. Group communication is one essential building block for the development of such an infrastructure. Traditionally, such systems were often built with focus on rather static systems. The advantage of such static settings is that accurate prediction and analysis is feasible, making them the ideal choice for critical applications with strict dependability requirements.

Today, an increasing number of computing systems are distributed, and an increasing portion of daily life is affected by such systems. Even simple applications (e.g., some Internet-based web service) are faced with fault-tolerance requirements. This has an important impact on the design of a fault-tolerance infrastructure: On the one hand, every application may have different requirements regarding fault-tolerance properties, security, responsiveness (reaction time and throughput), and so on. On the other hand, the environment may vary significantly regarding properties like the failure model (e.g., crash-stop, crash-recovery, and Byzantine), timing or synchrony aspects, or available communication means (e.g., 1 Gbit/s LAN vs. 19.2 kbit/s GSM mobile).

For the construction of a fault-tolerance infrastructure, this has two important impacts: First, best service quality will be obtained if the infrastructure allows *application- and environment-specific tailoring* depending on the requirements of the application and the properties of the environments. Second, the infrastructure needs to support *flexible run-time adaptation*, as both the needs of the application and the environment may change dynamically at runtime.

Let us, for example, consider the failure model: A crash-stop model, where processes either function correctly or have failed permanently, has the advantage of least complexity and minimal overhead, so it might have initially been chosen for some system. However, a crashed node can not simply recover and continue operation in this configuration.

Because of this disadvantage, the failure model might be adjusted to crash-recovery at some point. Typically, this requires some stable storage to keep critical state across crashes. Thus, it slightly increases the overhead of all operations, but nodes can seamlessly continue operation after recovery.

Finally, our distributed application might be confronted with changed security considerations that lead to the demand of intrusion tolerance. Such demand is satisfied by reconfiguring the system to support a Byzantine failure model, which allows to tolerate even malicious intrusions (as long as the number of faulty nodes is not too high, typically less than one third of all nodes).

In order to support such adaptive fault tolerance in our AspectIX middleware system [17], we were faced with the need of an adequate support for tailoring and run-time adaptation at the group-communication level. Existing systems for group communication could not meet our requirements regarding these issues. If run-time adaptation is supported in existing systems at all, it is typically limited to changing group membership or, in some cases, dynamically adjusting timing parameters of failure detector modules.

Our AspectIX group communication system (AGC) uses an encapsulated consensus module to obtain total order. This generic module allows many possible specializations and thus provides an ideal basis for application-specific tailoring. These specializations include the classic Paxos algorithm [13] and variants for low latency as well as for crash-stop, crash-recovery, and Byzantine failure models. As our system supports complete dynamic reconfiguration, it meets the requirement of flexible run-time adaptation.

This paper is structured as follows. Section 2 discusses related work on consensus algorithms and group communication. Section 3 presents the general architecture of our system. Section 4 describes our generic consensus implementation and its specializations. Section 5 discusses the implications of run-time reconfiguration in detail and Section 6 shows run-time measurements, which illustrate the performance of our Java implementation. Finally, Section 7 concludes.

## 2 Related Work

### 2.1 Distributed Consensus

In an asynchronous distributed system, deterministic consensus is impossible if at least one node fails; this fact is well-known as *FLP impossibility* [9]. Unfortunately, most distributed systems (e.g., those that use today's Internet as communication infrastructure) cannot count on synchrony. Consequently, all practical algorithms must be built on some *partially* synchronous model that is strong enough to avoid the FLP impossibility, but that is also weak enough to have realistic assumptions on the basic communication system.

Distributed consensus in systems without synchrony was probably first addressed by Lamport with the Paxos algorithm [13]. Subsequently, several authors provided further work on Paxos [2, 16], describing variants for crash-stop as well as for crash-recovery without stable storage. The idea of Brasileiro et al. [3] may also be applied to Paxos to obtain a fast (i.e., low-latency) variant. Castro presented variants of a practical consensus algorithm for the Byzantine failure model [6]. Due to structural similarities with the Paxos algorithm for the crash-recovery model, this algorithm is also referred to as "Byzantine Paxos".

Other authors have used different approaches to distributed consensus. Chandra & Toueg [7] introduced the idea of unreliable failure detectors to encapsulate the additional assumptions that are necessary to solve consensus in asynchronous systems. This allows using the same consensus algorithm with different, environment-specific implementations of the failure detector. The ABBA algorithm [4] works in a completely asynchronous system with Byzantine failures using randomization. However, these works do not focus on issues related to tailoring and reconfiguring service properties.

A few works are closely related to ours regarding the generalization of consensus for offering configurable variants with a generic interface. The General Agreement Framework (GAF) [11] bases on the algorithm of Chandra & Toueg and allows parameterization at instantiation time. It mainly allows to select predicates for considering nodes crashed or alive, for judging proposed values as acceptable, and for allowing early decisions. The Generic Consensus Service (GCS) [10] of Guerraoui and Schiper aims at providing a reusable service that allows solving various problems, including atomic commit, group membership, and group communication. DisCusS [5], a distributed consensus service, is based on self-adapting failure detectors, which allow optimizing the performance of consensus by reducing false suspicions of the failure detector. Some existing work addresses the question of adaption at the failure detectors level. Bertier et al. [1] analyze the effect on service quality that dynamically adjusting the frequency of periodic alive-messages and the timeout period achieves based on system monitoring.

However, all these systems use a fixed crash-stop (or crash-recovery) model and limit run-time adaptation to changing timing parameters. In contrast, our work addresses a broader scope of configurability: First, reconfiguration may affect the system model (including a Byzantine failure model), various struc-

tural variants for different optimization goals, and timing parameters. Second, all configuration may also be dynamically adjusted at run-time.

### 2.2 Group Communication

Group communication, or total-order multicast, has been addressed by many researches for more than two decades, as it is without doubt an essential mechanism for constructing reliable distributed systems. The survey of Défago, Schiper, and Urbán [8] gives an extensive overview about around 60 known group-communication systems.

One approach to total-order group communication, first proposed by Chandra & Toueg [7], is to transform this problem into the consensus problem. Several practical group communication systems use such approach and apply a modularization that builds upon some kind of consensus module. Mostefaoui and Raynal [15] describe a optimization that restricts the use of the consensus algorithm to situations where asynchrony and crashes prevent nodes from obtaining a simple agreement on message order. Eden [12] is a group communication system based on the above-mentioned Generic Agreement Framework [11]. Unfortunately, no performance characteristics or further details have been published. Larrea et al. [14] also used the Chandra & Toueg algorithm to implement totally-ordered broadcast and evaluated the resulting performance. Rodrigues and Raynal [18] apply the Chandra & Toueg transformation—which assumes a crash-stop failure model— to the crash-recovery model.

All described systems, however, have a slightly different focus than our system, as we place emphasis on a broader scope for configuration (e.g., crash-stop, crash-recovery, and Byzantine failure model) and on integrated support for flexible run-time reconfiguration.

## 3 Design of the Consensus-Based Reconfigurable Group Communication System

### 3.1 Overview

The group communication system has a modular design as shown in Figure 1. The core component of any communication group is `Group`. It implements the interface that is visible to the client. This interface offers methods to send and receive group messages, as well as to configure the group by adjusting group membership or group policies.

The `Consensus` component is used to obtain a total order of all group messages. A variety of implementations are available, each with different quality-of-service properties.

Both components use an instance of `ComSys`. `ComSys` encapsulates the low-level communication between all participating nodes. For this purpose, it offers unidirectional one-to-one and one-to-many communication, as well as network-independent addressing of nodes.

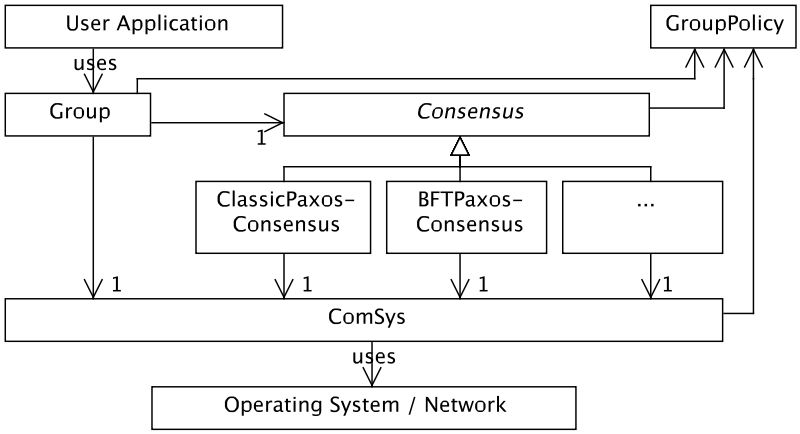


**Fig. 1.** Modular Structure of the AspectIX Group Communication System (AGC)

The configuration of all three main components is described by a `GroupPolicy`. This policy is defined at group creation time and may later be changed by the dynamic reconfiguration process.

### 3.2 `ComSys`: The Low-Level Communication System

The `ComSys` component has to fulfil a set of tasks: It represents an abstraction that encapsulates the specific mechanisms used for communication (e.g., TCP, TLS, SOAP/HTTP). It handles failures by queueing messages and re-establishing connections after failures. It supports a crash-recovery model by providing network-independent addressing, as a recovered node may, e.g., use a different dynamically assigned IP address or a different TCP port number. Finally, it offers a fully asynchronous (non-blocking) sending primitive to the other components. The internal structure of `ComSys` is outlined in Figure 2. The design supports multi-threaded use by the client, i.e., multiple client threads may simultaneously send or receive messages.

All internal sub-components of `Group` and `Consensus` share the same `ComSys` instance. Each message is tagged with a message type to allow a direct delivery to the appropriate entity. Each entity that accesses `ComSys` may obtain a `ComIdAdapter` object and may register a `MessageListener`. The `ComIdAdapter` automatically tags all outgoing messages with a specific message type. All incoming messages of a specific type are forwarded to the corresponding set of `MessageListener` instances.

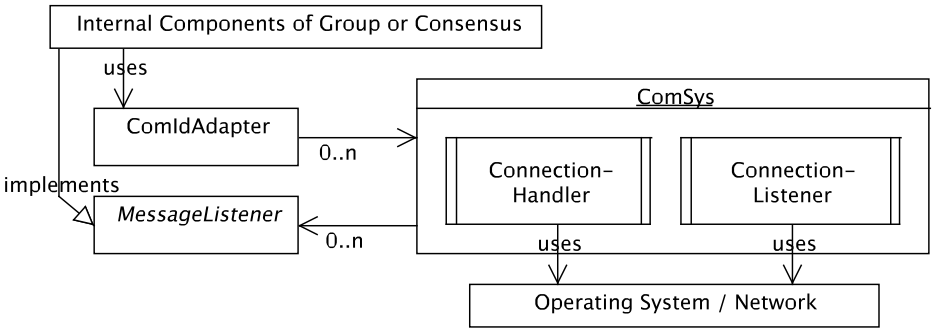


**Fig. 2.** Internal Structure of the Low-Level `ComSys`

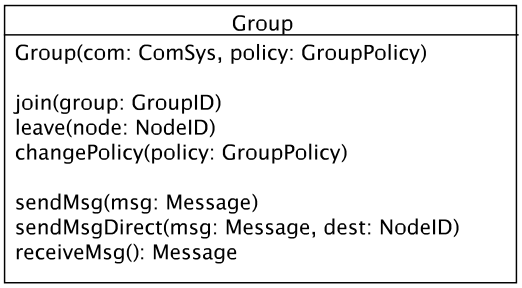


**Fig. 3.** Interface of the `Group` component

At the bottom end, `ComSys` internally uses two active threads: Outgoing messages are handled by the `ConnectionHandler`. It queues messages, sends messages to available destinations, and handles re-establishment of low-level connections after failures. The `ConnectionListener` is responsible for all in-going connections and forwards all received messages to the appropriate `MessageListener`.

The group policy allows configuring the `ComSys`. Besides the low-level channel type (currently TCP or TLS; extensions for, e.g., SOAP/HTML are easily added), it allows configuring type-specific parameters like whether to use available hardware multi-cast mechanisms or timing parameters for re-establishing low-level connections.

### 3.3 `Group`: The Core Component

`Group` is the principal module of any instance of the group communication system. It provides the upper-level interface for the application (see Figure 3). Methods are provided for sending and receiving group messages as well as for configuring the system. In our group model, we distinguish three classes of nodes:

– *Core group members* are responsible for determining total order; they automatically learn all group messages.
– *External listeners* do not participate in an ordering protocol; nevertheless, they may learn all group messages in the globally consistent order.
– *External senders* may send messages to the group without being part of the core group; they may also receive replies from the group in reaction to requests sent to the group via the group communication interface.

In our terminology, we use the term *group* to refer to the core group exclusively. External listeners and external senders are referred to as *external nodes* that interact with the group. Traditionally, many group communication systems use a *closed-group* model, where only group members may send group messages. We get such a closed-group model if we restrict all nodes to be core group members. Without this restriction, we support the more flexible *open-group* model [8], where interaction between external nodes and the core group is supported. The group policy defines which nodes are allowed to interact with a group.

For (re-)configuring the group, three methods are provided in the `Group` interface:

– `join(group: GroupID)` may be used for joining a given group `group`. A new group member first instantiates a `ComSys` and `Group` component. The `join` operation tries to send a join request as an external node to one of the group members. As soon as the client receives a positive reply, it instantiates a properly configured `Consensus` component. The policy to be used is contained in the reply from the group.
– `leave(node: NodeID)` requests that the given node is removed from the group. It is both possible that some node requests its own removal and that it requests the removal of another (e.g., permanently crashed) node.
– `changePolicy(policy: GroupPolicy)` is available for all other reconfiguration actions. All policy changes are subject to the group's agreement and are total-order delivered to all group members.

It is necessary to distinguish *soft* and *hard* reconfiguration requests. All requests are passed to the group via the group's total-order protocol. A *soft policy change* may simply be applied to all internal components of the group communication system at some node, as soon as the new policy is received. Such changes may, for example, affect timing parameters of a failure detector. A *hard policy change* needs additional coordination to ensure a safe transition to the new configuration. One example for this case is the complete replacement of the `Consensus` module.

A policy may further restrict acceptable operations by imposing limitations on valid policies, permitted senders, etc. If a operation is not accepted, consensus will decide the rejection of that operation. More details of the policy-based reconfiguration process will be discussed in Section 5.

The internal behavior of `Group` differs between external nodes and core group members. A core group member has a `Consensus` component, and `Group` basically passes all application requests (messages to be sent as well as all kind of

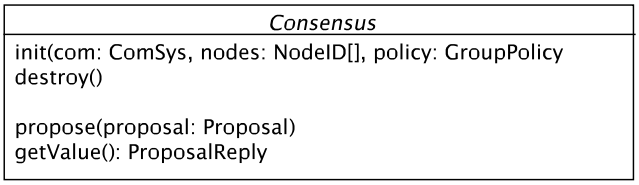


**Fig. 4.** Interface of the `Consensus` component

reconfigurations) as *consensus proposals* directly to its `Consensus` module. An external node has no `Consensus` module. Instead, it forwards all client requests as simple direct messages to a core group node. This group node then propagates this request to the group.

Another group policy influences the behavior of an external node. With the default *send-to-one* policy, such a node sends its request to one of the group members; if available, a primary or "leader" node is selected as recipient. For low-latency consensus algorithms based on the idea of "consensus in one communication step" [3], all nodes participating in the consensus protocol need to know the initial value. Thus, the sender has to broadcast its message to the whole group, which is specified with a *send-to-all* policy. A third policy, *send-to-one-retry-all* first sends the message to one group node. If the message reception is not acknowledged by the group within a specified time, it is re-sent to the whole group. This procedure may, e.g., be used with Castro's algorithm [6] for Byzantine failures. It is an optimistic approach that uses a minimal number of messages in the good case (the selected contact node is not faulty). If it is faulty, re-sending the message to all nodes ensures that it will eventually be delivered to all.

### 3.4 `Consensus`: Using Consensus for Total Order

It is the task of a total-order protocol to define an order on all messages sent to the group. In our system, a fault-tolerant consensus algorithm is used to define this order. Basically, this means that each message to be delivered is subject to a consensus decision.

The generic interface to `Consensus` is shown in Figure 4. It is only used by `Group` and not directly visible to the application. The `propose` operation passes a *proposal* as input for a consensus instance. The `getValue` method blocks until the next consensus instance reaches a decision and returns the decision result. Two additional methods are provided for initialization and clean shutdown. The semantics of the operations are defined as follows: The `propose` method has to guarantee that the proposal is eventually decided upon by the consensus instance, as long as the node that initiated that proposal does not fail. The

`getValue` method returns values that have been decided in a globally defined order.

Using one consensus decision for each group message is a bottleneck. Therefore, a batching mechanism may be used: The group collects all messages sent to the group during some (short) period of time. One consensus decision defines the order of all such messages. This significantly reduces the overhead caused by the consensus algorithm; it however increases the latency slightly by the interval in which the system waits to collect messages. The run-time measurements in Section 6 illustrate the effect of such a batching mechanism. In the next section, we will focus on details of the consensus implementation.

## 4 Generic Paxos-Based Consensus

The consensus component of our group communication system is designed to be generic. Any agreement implementation that provides the above interface (as described in Section 3.4, Figure 4) can be used. Our current prototype implements a generic model for several variants of the Paxos algorithm. These variants include the classic Paxos using stable storage for crash-recovery [13], variants for crash-stop and crash-recovery without stable storage [2], and for Byzantine failures [6]. Specializations allow to optimize for low latency or for minimal communication costs.

For group communication, multiple instances of consensus, numbered consecutively by *instance numbers*[3], are necessary; each instance corresponds to deciding the delivery of one message or one batch of messages.

Multiple consensus attempts may be executed for one consensus instance. We refer to these attempts as *rounds*[4]. A total order exists on all round numbers. The Paxos algorithm ensures that a decision in round $i$ is never inconsistent with a previous decision of the same consensus instance in some round $j < i$.

### 4.1 Classic Paxos

The Paxos algorithm works in three phases. Each instance for a single decision may be considered as a 3-phase commit protocol, where the value to be committed is not yet known in the first phase. Instead, Phase 1 collects information about values that have potentially been committed in previous rounds. Phase 2 sends a proposal to the group. This is either the value learned in the first phase (which ensures consistency with previous consensus attempts), or, if no such value exists, an externally provided value. If sufficiently many nodes acknowledge the reception of the proposal, it may be committed in Phase 3.

[3] Unfortunately, authors writing about Paxos tend to use differing terminology. The term *instance numbers* is consistent with De Prisco; they are called *decrees* in Lamport's original work. In Castro's algorithm, they correspond to *sequence numbers*.

[4] The term *round number* is again consistent with De Prisco. Lamport refers to these rounds as *ballots*; Castro uses the term *view*.

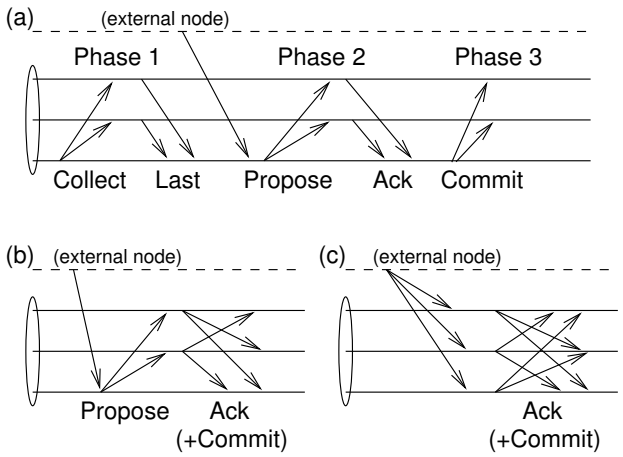


**Fig. 5.** Speed Variants of the Generic Paxos Implementation

Typically, the first phase is only executed when starting a new consensus attempt, e.g., after a leader change. As a further optimization, the first phase may be executed jointly for all instances: The leader sends a collect query to all others, indicating the lowest instance number whose decision result it does not know. If this request does not have the highest round number, it will be rejected. Otherwise, all nodes reply with a last message containing a list of all proposals or decisions known to them in higher-numbered instances.

Classic Paxos, as described by Lamport in [13], has a typical message exchange pattern as shown in Figure 5a. After the first phase, it requires three message delays for each consensus decision (Propose, Acknowledgement, Commit). Embedded in the group communication system, usually one additional message delay arises from the necessity to send the proposal to the leader node. Such a proposal may be sent either from a non-leader node participating in the consensus, or, as shown in Figure 5, from an external node.

### 4.2 Paxos Speed Variants

The interaction patterns of non-Byzantine speed variants are shown in Figure 5b,c (Phase 1 is omitted, as it is identical in all variants). In the Fast Paxos variant (b), the acknowledgement and commit messages are joined by broadcasting the acknowledgement to all group nodes, which in turn may decide autonomously if sufficient acknowledgements have been sent. This variant essentially improves latency at the cost of an increased number of messages to be sent.

The Ultra Fast Paxos variant (c) uses the idea of one-communication-step consensus [3]. It is assumed that all group nodes have the same initial value (which requires a proposal to be broadcast to all nodes instead of sending it only

to the leader). If all nodes receive identical acknowledgments, they may commit immediately. If not—which is easily the case when several clients try the propose values concurrently—a conflict is detected, and the algorithm reverts to classic Paxos. Consequently, this variant improves latency even further in an optimistic case. Furthermore, the crash of the designated leader has no negative influence on consensus (while in the other variants, a required leader change adds an additional delay); These improvements come at the cost of worse performance when concurrent access occurs.

### 4.3 Handling Crash-Recovery Failures

In a crash-stop system model, all variants may be implemented in a straightforward way, as all nodes either function correctly or crash permanently. In a crash-recovery model, a correct node may temporarily crash and subsequently recover again, continuing to participate in the consensus protocol.

To make this continuation possible without provoking inconsistencies, significant state information must not be lost in a crash-recovery step. For example, in Phase 1 of the Paxos algorithm, each node has to send information about any value previously accepted by an acknowledgement, even if this acknowledgment precedes a crash-recovery cycle; thus, prior to sending such a acknowledgement, the received proposal needs to be written to some kind of stable storage.

A crash-recovery model is supported in all Paxos variants by using such a stable storage. This stable storage may, for example, be implemented using flash memory, a hard disk, or redundant hard disks, depending on available hardware and on the kind of physical faults to be tolerated. The implementation has to make sure, that a crash during write operations to stable storage does not lead to an inconsistent state. Our prototype implementation uses a hard disk to store state. After recovery, the internal state of the consensus algorithm can be recovered from stable storage.

### 4.4 Handling Byzantine Failures

The basic structure of Paxos is also present in Castro's agreement algorithm [6] for Byzantine failures. Its interaction pattern requires three communication steps normal-case operation (i.e., in the second and third phase). Our implementation currently only supports the public-key based approach. Castro's variant without public-key cryptography in normal operation would allow to increase to performance, especially with a small number of participating nodes.

### 4.5 Parallel Execution of Consensus

Using consensus for group communication, a sequence of consensus instances is executed, each labeled uniquely by a continuous *instance number*. As these instances are generally independent from each other (we will discuss restrictions in this issue in the next section), they can be executed in parallel.

| | | |
|---|---|---|
| ComSys:Type | hard | TCP/IP-Communication |
| ComSys:Reconnect | soft | 60s |
| ComSys:Encryption | hard | no |
| ComSys:Multicast | soft | no |
| Consensus:Type | hard | PaxosAgreement |
| Consensus:Mode | hard | StableStorageRecovery |
| Consensus:Timeout | soft | 10s |
| Consensus:ParallelInstances | soft | 5 |
| Group:BatchDelay | soft | 100ms |
| ... | ... | ... |

**Fig. 6.** Policies for Configuring the Group Communication System

There is a some benefit from such parallelism, as the delay between successive decisions is substantially reduced. Furthermore, such a parallelism allows to join messages at low-level communication; e.g. a *Propose* message of one consensus instance can be transitted in combination with a *Commit* message of the previous instance. We will discuss the practical impact in Section 6.

### 4.6 Generic Implementation

The structural similarity of all described variants allows a generic implementation strategy, which is less error-prone and simpler than implementing a consensus module for each variant from scratch. Our prototype uses an abstract base class that encapsulates the general Paxos logic (processing the three phases for each instance, handling leader changes, etc.). Specializations add all elements that are unique to either the crash-failure or the Byzantine-failure variant. Stable storage or fast and ultra-fast configurations only require minimal additional logic in the implementation.

## 5 Policy-based Reconfiguration

### 5.1 Overview

The complete configuration of our group communication system is controlled by the group policy. This policy is represented by a key/value-map; Figure 6 shows typical values. All members of a group have the same policy, which is initially defined at group creation time. A joining node is automatically informed about the currently valid policy. This section discusses how various policy elements can be reconfigured dynamically at runtime, and what support is therefore needed in the implementations of `Group` and `Consensus`.

### 5.2 Performing Consistent Reconfiguration

All reconfigurations need to be performed consistently by the whole group. For this purpose, each reconfiguration is sent to the group as consensus proposal.

The consensus decision not only defines the new policy to be adopted, but also determines exactly at which instance number this change is to be made.

Most policy changes, which we classify as "soft", may simply be applied to all components as soon as they are decided by the group. This kind of policy change is fully transparent to the application using the group communication system. The only run-time cost is, that one consensus instance needs to be executed to decide the new policy. This decision can be bundled with decisions on application messages, with the same bundling mechanism as described in Section 3.4. Thus, the cost for such reconfigurations is basically reduced to the mere effort of sending the policy to all group nodes.

### 5.3 Handling "Late" Nodes

Due to the asynchrony of the system model and the ability to tolerate faults (i.e., to decide the order of message delivery without the participation of all group nodes), some nodes might already have finished executing the consensus instance $i$ (and maybe even subsequent instances $i' > i$), while others have not. This is particularly a problem for reconfigurations like exchanging the consensus instance. Delaying the reconfiguration until all group nodes have finished the concerned consensus instances is not a viable option, as this will severely hinder reconfiguration if just one node is unavailable.

Two solutions are possible: Either old consensus instances have to be kept active until all group nodes know the decision value, or successful decision results have to be managed by a component that is always available in the system. In our system, we use the second option, as it simplifies internal management and consumes less resources. As soon as consensus is reached in one instance, the result is managed by the `Group`, and the consensus instance may be discarded. If a node lacks one decision result of one instance number and the corresponding consensus instance is no longer available, `Group` responds directly with an update message containing the final decision result. Furthermore, `Group` contains a garbage collection mechanism: Decision results are kept in a decision log only until each group node either has acknowledged the reception of that decision or has crashed permanently.

### 5.4 Reconfigurations and Parallel Instances

Section 4.5 explained, that multiple consensus instances may be executed in parallel, as they are independent of each other. This strategy contributes significantly to the system performance. However, the possibility of dynamic reconfigurations has an important impact on such a parallelism:. Suppose that a reconfiguration gets decided in consensus instance $i$. If this reconfiguration is supposed to influence instance $i + 1$ (e.g., select which algorithm to use), we must not start instance $i + 1$ until $i$ is decided. Consequently, no parallelism is possible.

The solution to this problem is to limit the parallelism and delay the validity of reconfigurations. In this scheme, a reconfiguration decision in instance $i$ gets

valid only for consensus instances with a number greater than or equal to $i + N$, with $N$ being a constant defined by a policy. This way, up to $N$ consensus instances can be executed simultaneously. In practice, it is not essential to execute an infinite amount of parallel rounds; just a small number is needed. The drawback is, that a reconfiguration is delayed for $N$ consensus executions. This may result in a long delay if no application messages are sent. To avoid this problem, a sequence of $N$ no-operation proposals can be proposed to the group. This sort of strategy has also briefly been mentioned by Lamport in [13].

### 5.5 Handling Hard Reconfigurations

In contrast to *soft* reconfigurations, which are handled as described in Section 5.2, a *hard* reconfiguration is one that leads to an incompatible system modification. This is the case for modifications of the `consensus` module (replacing the algorithm) or of the `ComSys` (e.g., switching from plain TCP to SSL encryption).

The `group` module performs a clean change-over at a determined instance number $i$. It waits for completion of all instances less than $i$, initializes the new module implementation, transfers all relevant state information of the old implementation, and finally activates the new module.

### 5.6 Membership Changes

Membership changes by join or leave operations may be considered as soft or as hard. Treating them as soft, it is essential that the consensus implementation is able to handle membership changes internally. Treating them as hard removes this requirement, but increases the cost of the reconfiguration. Unfortunately, some consensus implementations do not support such a internal reconfiguration; for example, the view-change mechanisms of Castro's algorithm assumes a static number of nodes. In such a case, a hard reconfiguration is performed by replacing `Consensus` with a new instance having the same type, but a different node set. On the other hand, node set changes can easily be integrated, e.g., in a implementation of classic Paxos. In cases like this, a soft reconfiguration strategy is to be preferred. Our implementation supports both variants to obtain best efficiency for hard reconfiguration steps without limiting possible implementations of the consensus module.

## 6 Validation

We have carried out several test to evaluate to performance of our consensus-based group communication system. All test have been done on GHz Intel Pentium 4 (3.0 GHz) workstations running Linux (kernel 2.4.30), connected via a 100-BaseT network.

The most important factor in system performance of a consensus-based group communication system is the cost for a consensus decision. Figure 7 shows the

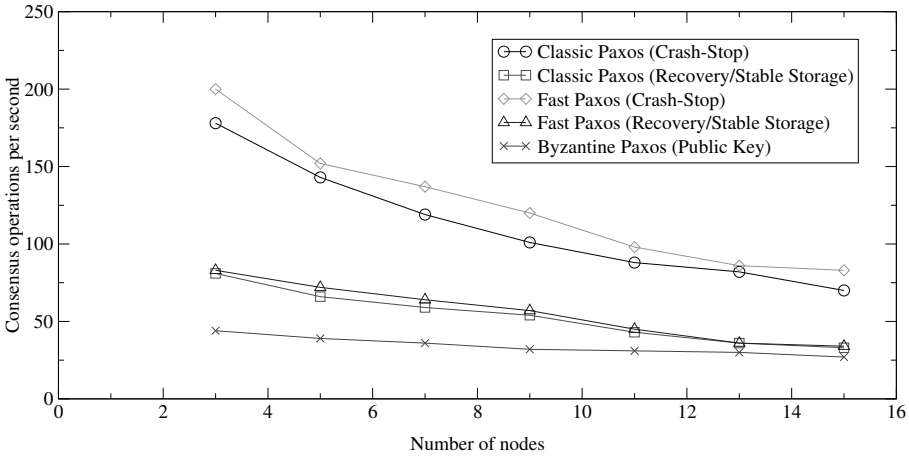


**Fig. 7.** Consensus Operations per Second

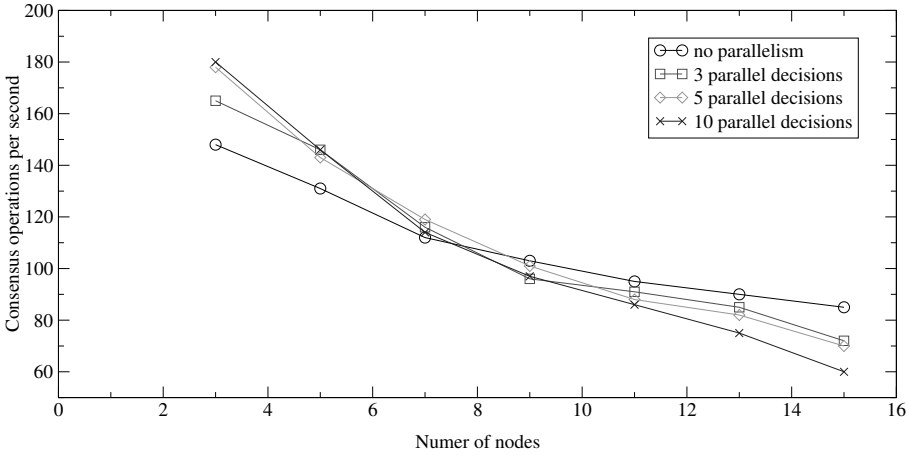


**Fig. 8.** Effect of Parallelism on Classic Paxos (Crash-Stop)

number of consensus decisions per second that our system achieves in relation to the number of core group nodes, for various consensus instances. The recovery variants use synchronous writes to the local disk as stable storage; the parallelism of consensus decision was limited to five parallel instances.

For all variants, the system scales well with an increasing number of nodes. The limiting factor in the stable-storage variants are the synchronous write operations; thus there is only little difference between Classic Paxos and Fast Paxos in these cases. The Byzantine consensus instance is, as it might be expected, the most costly variant. Our current prototype uses public-key based signatures; we do not yet support the more efficient variant of Castro [6] based on symmetric message authenticators.

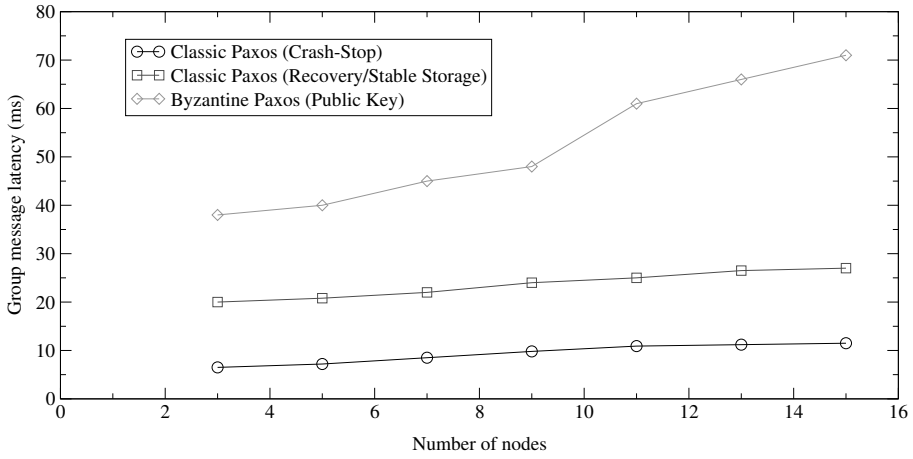


**Fig. 9.** Over-all Group Message Latency with Different Consensus Algorithms

Parallelism of consensus operations, as explained in Section 4.5, makes reconfiguration a slightly more complicated task. Therefore, we examined in another experiment the benefit of such parallelism. Figure 8 shows the number of consensus decisions per second for different degrees of parallelism. For a small number of nodes (less than nine), such parallelism increases the performance. Somewhat unexpected, the performance decreases at a higher number of nodes, which is probably due to the overhead of internal synchronization. The results thus also show that a dynamic configuration of the parallelism depending on the number of nodes is necessary to get optimal performance. Hardly any difference exists between five and ten parallel rounds, which coincides with the exception that a small number of parallel rounds is always sufficient.

From the application point of view, an essential parameter is the message latency. Figure 9 shows the latency for three different consensus variants, depending on the core group size. All times are per-message latencies averaged over 100 messages sent to group, measured at the application-level group interface.

## 7 Summary and Future Work

We have presented a group communication system based on fault-tolerant consensus algorithms. Our system makes two main scientific contributions: First, it allows to be tailored to application-specific and environment-specific requirements. The broad range of these customizations includes, among others, the failure model (crash-stop, crash-recovery, and Byzantine), low-level communication mechanisms, and timing properties. Second, it efficiently supports run-time reconfiguration of all such customizations without service interruption.

The design of our group communication system, composed of the `Group`, a `ComSys`, a `Consensus`, and a `GroupPolicy` instance was presented in detail. The consensus instance encapsulates arbitrary consensus algorithms; our current implementations uses variants of the Paxos algorithm. We have given an detailed overview of these variants, which support several failure models and parameterizations to optimize latency and message overhead. The policy-based configuration mechanisms allows run-time reconfiguration transparent to the application and with negligible overhead. A practical performance analysis of our implementation evaluated our system design. We have analyzed the throughput and latency characteristics of different configurations to illustrate the feasibility of our approach.

Our current prototype has not yet been optimized, so we still anticipate further improvements of the presented measurements. Adding additional variants of the agreement component, like Byzantine consensus based on symmetric message authenticators or the ABBA algorithm, is being considered. On a broader scope, we currently work on a middleware integration of our system; we target, first, at a integration into the FT-CORBA implementation GroupPac for active replication; second, we will use the system in our dynamically configurable AspectIX middleware.

## References


1. Martin Bertier, Olivier Marin, and Pierre Sens. Implementation and performance evaluation of an adaptable failure detector. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'2002)*, pages 354–363, 2002.
2. Romain Boichat, Partha Dutta, Svend Frølund, and Rachid Guerraoui. Deconstructing Paxos. *SIGACT News*, 34(1):47–67, 2003.
3. Francisco Brasileiro, Fabíola Greve, Achour Mostéfaoui, and Michel Raynal. Consensus in one communication step. In *Proc. of Parallel Computing Technologies, 6th International Conference, PaCT 2001, Novosibirsk, Russia, September 3-7, 2001*, pages 42–50, 2001.
4. Christian Cachin, Klaus Kursawe, and Victor Shoup. Random oracles in constantipole: practical asynchronous byzantine agreement using cryptography (extended abstract). In *Symposium on Principles of Distributed Computing*, pages 123–132, 2000.
5. Lásaro J. Camargos and Edmundo R. M. Madeira. DisCusS and FuSe: considering modularity, genericness and adaptation in the development of consensus and fault detection services. In Rogério de Lemos, Taisy Weber, and João Batista Camargo Jr., editors, *Proceedings of the Latin American Symposium on Dependable Computing (LADC 2003)*, volume 2847 / 2003 of *Lecture Notes in Computer Science (LNCS)*, pages 234 – 253, São Paulo, SP – Brazil, October 2003. Springer-Verlag Heidelberg. ISSN: 0302-9743.
6. Miguel Castro. *Practical Byzantine Fault Tolerance*. PhD thesis, MIT Laboratory for Computer Science, Cambridge, MA, 2001.
7. Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.
8. Xavier Défago, André Schiper, and Péter Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4):372–421, 2004.

9. Michael J. Fischer, Nancy Lynch, and Michal S. Paterson. Impossibility of distributed consensus with one faulty processor. *Journal of the ACM*, 32(2):374–382, 1985.
10. Rachid Guerraoui and André Schiper. The generic consensus service. *IEEE Trans. Softw. Eng.*, 27(1):29–41, 2001.
11. Michel Hurfin, Raimundo A. Macêdo, Michel Raynal, and Frederic Tronel. A general framework to solve agreement problems. In *Symposium on Reliable Distributed Systems*, pages 56–65, 1999.
12. Michel Hurfin, Jean-Pierre Le Narzul, Xiaojun Ma, and Frédéric Tronel. Eden : a group communication service. `http://www.inria.fr/rapportsactivite/RA2003/adept2003/module9.html`.
13. Leslie Lamport. The part-time parliament. Technical Report 49, System Research Center, Digital Equipment Corp., Palo Alto, September 1989.
14. Mikel Larrea, Alberto Lafuente, and Cristian Martín. A modular middleware for reliable distributed programming. In *IADIS International Conference Applied Computing, Algarve, Portugal*, pages 161–166, 2005.
15. Achour Mostefaoui and Michel Raynal. Low cost consensus-based atomic broadcast. In *PRDC '00: Proceedings of the 2000 Pacific Rim International Symposium on Dependable Computing*, page 45, Washington, DC, USA, 2000. IEEE Computer Society.
16. Roberto De Prisco, Butler W. Lampson, and Nancy A. Lynch. Revisiting the Paxos algorithm. In *Workshop on Distributed Algorithms*, pages 111–125, 1997.
17. Hans P. Reiser, Franz J. Hauch, Rüdiger Kapitza, and Andreas I. Schmied. Integrating fragmented objects into a CORBA environment. In *Proc. of the Net.ObjectDays (Erfurt, Germany)*, 2003.
18. Luís Rodrigues and Michel Raynal. Atomic broadcast in asynchronous crash-recovery distributed systems. In *ICDCS '00: Proceedings of the The 20th International Conference on Distributed Computing Systems ( ICDCS 2000)*, pages 288–295, Washington, DC, USA, 2000. IEEE Computer Society.