# MojaveComm: A View-Oriented Group Communication Protocol with Support for Virtual Synchrony

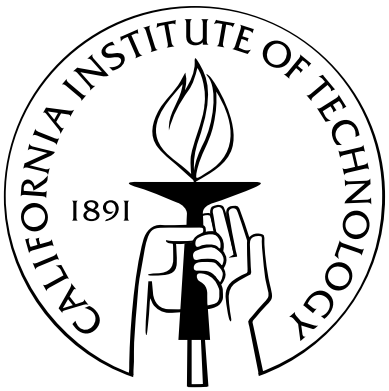
Thesis by

David A. Noblet

In Partial Fulfillment of the Requirements

for the Degree of

Master of Science




California Institute of Technology

Pasadena, California

2008

(Submitted June 4, 2008)

To my supportive family and friends.

# Acknowledgements

I would like to extend gratitude to my advisor, Professor Jason Hickey, and to all the members of the Mojave Lab, past and present, who took the time to participate in all those lively whiteboard discussions. In particular, I want to offer my special thanks to Cristian Ţăpuş, Nathaniel Gray, Mihai Florian, and Joshua Goldstein for all of their insightful input and contributions to this work.

# Abstract

In this thesis, we explore the feasibility of implementing a general communication protocol that addresses common classes of problems that one encounters in the development of distributed applications, such as: multipoint-to-multipoint communication, message (re)ordering, mutual exclusion, and consensus. The paper details both the design and implementation of MojaveComm, a view-oriented total-order group communication protocol suitable for deployment on wide-area networks. Moreover, we provide a high-level overview of MojaveFS, a sequentially consistent distributed filesystem, and show how we can use the message-ordering guarantees of MojaveComm as the basis for the implementation of its sequential consistency guarantees.

# Contents

# List of Figures

# Chapter 1

# Introduction

In recent years, there has been a proliferation of network-enabled applications. In large part, this is due to the advent of cheap and readily-available access to global communications infrastructure provided by the Internet. Nevertheless, this physical access is only one contributing factor to the momentum of this trend. Another influence, perhaps more important, is the success of the communication protocols that run on top of this physical infrastructure: namely, the TCP-IP protocol suite.

The Transmission Control Protocol (TCP [19]) has supported rapid expansion of network-enabled services and applications because it significantly decreases the barrier to entry for developers. TCP provides a concise set of strong communication guarantees upon which one can build more complicated application-specific communication protocols to suit the needs of a specific service. Moreover, these assurances provided by TCP represent a large common subset of the functionality required by many applications while still maintaining a simple semantics that is easy to reason about (and build upon).

As a testament to the ubiquity of TCP, many of the most popular network-enabled applications and services make use of this protocol: web browsing (HTTP), email (POP, IMAP, and SMTP), file transfer (FTP), remote login (telnet, ssh), chat (IRC), etc. Nevertheless, TCP is not ideally suited for all applications. In particular, TCP provides only point-to-point communication between nodes. In cases where data needs to be disseminated to multiple entities simultaneously, the use of TCP may be inefficient – or the guarantees it provides may be insufficient to directly meet the needs of the application.

This fundamental limitation of TCP has become more obvious as the push to develop larger and more complex network-enabled systems moves forward. Specifically, the recent interest in *distributed systems* by the software community has worked to move this issue to the forefront. Distributed systems are a natural extension of traditional network-enabled applications. Traditional network applications generally follow the client-server model, whereby some fixed set of network entities provide service (called servers) to a dynamic set of network entities that request service from the

former (called clients). When a client needs to access some service, it connects directly to one of the corresponding servers that provide that service and the server is then responsible for responding to the request of the client. Customarily, the server with which the client is communicating is a single network entity; in a distributed system, this is not necessarily true.

In the development of a distributed system, one seeks to provide a single service among multiple network entities. These network entities then act as a coherent unit, in spite of the fact that this unit is composed of distinct entities. Of course, in a distributed environment such as this, each individual component of the system must be able to make local decisions that somehow contribute to the service as a whole. Thus, in order to make informed decisions, the components must communicate with one another (over the network).

By taking a distributed approach to the solution of many problems, it is possible to eliminate many of the bottlenecks that are present in traditional centralized systems. The reasons for distributing a service over multiple distinct network entities are primarily threefold:

- *Reliability* - Through replication/redundancy, it is possible to ensure that the failure of one component will not disrupt the entire system.
- *Scalability* - As demand for the service increases, it is possible to add more components dynamically to meet this need.
- *Performance* - The more components that contribute to the service, the faster the service can respond (if the service is sufficiently parallelizable).

In fact, with current technology, many problems require a distributed solution to achieve the desired scale and performance.

Of course, once one starts to distribute the processing of some service across multiple entities, it becomes necessary to consider issues such as data consistency and communication loss/error. And, to make matters worse, when such communication between entities is over unreliable communication channels, one runs up against fundamental impossibility results regarding consensus in this context [10]. In the end, it is these types of problems that must be addressed when developing a distributed network application. And, often times, the solutions to these issues greatly complicates the design and implementation of such protocols. Nevertheless, many of the problems that one encounters in the design of a distributed application or service are not specific to the particular application in question, but are rather more general problems inherent in the design of such distributed systems. For example, typical classes of problems include: multipoint-to-multipoint communication, message (re)ordering, mutual exclusion, and consensus.

In this thesis, we address the question of whether it is possible to implement a general communication protocol that solves these common distributed systems problems, providing evidence in

support of an answer in the affirmative. In the following text, we present the design and implementation of *MojaveComm*, a group communication protocol intended to allow groups of network entities (called nodes) to self-organize into groups within which MojaveComm provides strong ordering guarantees with respect to the transmission and receipt of messages; the specification of this protocol is given in Chapter 2 and an overview of the implementation is detailed in Chapter 3. And Chapter 4 is a case study illustrating an application of MojaveComm to a particular domain (in this case, distributed filesystems). Finally, we provide some closing remarks in Chapter 5.

## 1.1 Related Work

There is an extensive body of work in the area of reliable group communication systems that dates back to the 1980's. In early projects, like ISIS [5] and the process group extension to the V Kernel [7], the emphasis on the respective group communication protocol is secondary to that of a larger system. Nevertheless, many of these projects offer insight and develop abstractions that subsequent projects on group communication systems will borrow (such as in [6] [9] [4] [8] [17]). For example, nearly all of the works cited in this section provide some type of group membership abstraction; moreover, these projects also have at least limited support for state synchronization adhering to some flavor or other of the *virtual synchrony* model. In the text below, we provide an overview of the history of group communication systems (though this account is not exhaustive by any account).

The *process group* [7] extension to the V Kernel is among the earliest work in the area of group communication systems. The V Kernel is designed to be a distributed kernel that transparently coordinates multiple participating machines and presents them to the user as a single coherent system. The extension described in the paper introduces the process group abstraction as a mechanism whereby groups of processes may communicate with one another without prior explicit knowledge of the group membership. A process group is distinguished by its identifier (simply an integer in the implementation). In terms of governing group membership, a processes may *create*, *join*, or *leave* a group. Processes exchange messages in the context of a group using *send* and *recv*. This type of message exchange is considered to be reliable[1].

The ISIS [5] project introduces an abstraction called a *resilient object*. Conceptually, a resilient object is a record of data and operations that is replicated across multiple machines on a network. The replicas of a resilient object act as a coherent entity. In particular, operations performed on a resilient object are transactional in nature; either the operation is performed successfully such that the result consistent across all the replicas, or the operation fails (and the result is not reflected on any of the replicas). In its implementation, ISIS relies on a reliable, ordered message

[1] Although process groups in the V Kernel support reliable message delivery to members of the process group, it is important to note that the definition of *reliable delivery* in this context refers to the case when a message is successfully delivered to at least one member of the group (for the delivery to be considered a success, that is).

broadcast mechanism, *GBCAST* (for Group Broadcast), to ensure that replicas remain consistent. In GBCAST, messages are sent within the context of a *group* (different groups correspond to individual resilient objects in ISIS); each group has exactly one membership set and all messages sent to a group using GBCAST are totally ordered within that set.

Consul [16] is a system that aims to provide the user with a set of basic abstractions that can be used to build reliable systems based on the replicated state machine approach. Collectively, these abstractions embody three services: multicast, membership, and recovery. In Consul terminology, to send a message a process participates in a *conversation* with a set of processes; actual message exchange is handled by the Psync protocol. During the lifetime of a conversation, Psync maintains a "context graph" that satisfies the *happens-before* relation for all the messages exchanged in the conversation. To ensure correct ordering on delivery, each message is paired with a list identifying its dependencies calculated from this context graph. The membership protocol is responsible for the detection of remote process failures for each conversation the local process participates in. In the event of a failure (or detection thereof), the membership of the conversation is renegotiated and the recovery protocol takes over to ensure that all processes have received the same set of messages. As failure detection in Consul is conservative, it is possible that after a failure two or more distinct membership sets may exist for a single conversation; Consul does not provide any special facilities to handle this situation.

The VS [8] system provides explicit support for such *partitionable* group communication services (whereby membership of the communicating set may partition and recombine arbitrarily). As such, VS provides a group abstraction, similar to ISIS groups or Consul conversations, called a *view*. Although, views differ from the aforementioned abstractions in that a view is more like a partition or fragment of a group. Also, VS does not provide explicit recovery (i.e. message synchronization) on changes in view membership; rather, it simply exposes enough information such that it is possible to perform such synchronization at the upper layers if desired. An extension of VS, called DVS [20], broadens support for such group partitions by introducing the abstraction of *primary* views to VS. A primary view is usually considered to be the view that contains a majority of the available processes in the system; but generally speaking, the definition of primary simply needs to ensure that there is never more than one primary view at a time. The motivation is that, in spite of the fact many views may exist, to maintain their consistency guarantees many distributed applications require that at most one of these views be allowed to make progress.

The successors to ISIS, Horus and Ensemble [6], were an architectural and design effort to break from the monolithic and single-purposed origin of their predecessor in an attempt to provide greater flexibility and increased confidence in the correctness of the protocols. In both of these projects, the drive specifically was to develop a general framework for providing group communication services. The architecture was such that, although the interface is fixed, the specification can be satisfied

many ways in the implementation via the composition of different sets of *micro-protocols*. Because of correctness and performance concerns, Ensemble emerged as a reimplementation of Horus using the OCaml programming language; this was done to ensure that the developers could reason about the behavior of Ensemble using the automated proof assistant NuPRL. With NuPRL, it was possible to perform provably-correct code transformations to reduce overhead introduced by the many layers of micro-protocols.

Newtop [9] is a group communication system that, in addition to strict total ordering guarantees on message delivery, also provides explicit support for multiple groups and multiple views within each group. In Newtop, a process creates a group by specifying its maximal member set. Thus, initially all processes in a group have the same view membership. As there are network partitions and/or process failures, Newtop's failure detection mechanism renegotiates a new view with the other members of the existing view in order to agree on a new view that is a strict subset of the existing view membership. To join old members of a view, the process creates an entirely new group with the desired membership.

The Spread [4] group communication system is specifically designed for wide-area deployment. In order to remove the constraints placed on such a system by the diversity present in the physical network, Spread constructs an overlay network upon which it exchanges messages. In order to provide an efficient routing solution on top of this overlay, Spread introduces its own routing protocol, called Hop. Although, in practice this overlay serves to increase performance and efficiency of the protocol in general, Spread relies on a static configuration file to describe the underlying physical network topology on top of which the overlay is installed.

More recently, work on the QSM [17] project addresses the scalability problems seemingly inherent with group communication systems. The key insight in this work is that it is possible to take advantage of the potential overlap in separate group membership in order to deliver messages more efficiently (i.e. without having to filter out as many of the unwanted messages). In particular, the authors suggest a more efficient scheme to map application-level ("lightweight") groups onto low-level multicast ("heavyweight") groups. The authors claim scalability on the order of hundreds or thousands of participants for QSM using the scheme they present.

It is also worth mentioning work such as that found in [12], which attempts to make the ordering of messages between groups more efficient by constructing a *propagation graph* instead of using the traditional token-based protocols.

## 1.2 Overview

To put the project described in this paper into the context of the existing work, MojaveComm is a partitionable view-oriented total-order group communication system designed for wide-area

deployment. In particular, in this work we strive to provide a highly distributed and dynamically-configurable solution. For example, we avoid the use of a statically configured overlay network such as the one used in [4]. Moreover, we try to embrace the reliability constraints imposed on our system by targeting a wide-area network environment; to this end, we allow for group partitions to exist (and even provide mechanisms to facilitate their detection and recovery).

In our express formulation of the VIRTUAL SYNCHRONY property (see Equation 2.1), it is possible for many different sequencing and delivery policies to satisfy the specification. For example, because of the conditional nature of our guarantee, it is possible for the MojaveComm implementation to fine-tune its guarantees while still maintaining a consistent interface to the application-layer. Nevertheless, in spite of this flexibility, we want to be able to provide a more rigid and well-defined interface than that of [6] in order to avoid the problems of having an unconstrained protocol stack (and the introduction of the resulting optimization problem).

Also, unlike [8], we opt to embed our message synchronization mechanism and corresponding guarantees into the group communication system itself as we believe this makes applications implemented on top of MojaveComm easier to reason about (and, thus, eases the burden on the developer). And, although we recognize the utility of a system that natively supports a *primary view* abstraction (like [20]), we choose to defer this processing to the application. Even though MojaveComm does not support primary views directly, we ensure that the application-layer interface is sufficiently expressive to allow a wide range of application-specific primary view implementations on top of our basic service.

# Chapter 2

# MojaveComm Protocol Specification

This chapter presents the abstract behavioral specification for a network communication protocol called MojaveComm. This protocol is designed to provide strict message ordering guarantees between sets of communicating network entities (called nodes).

MojaveComm makes weak assumptions regarding the underlying communication mechanism upon which the protocol is built. The communication model for underlying message transport is as follows: messages may take an arbitrarily long time in transit; messages can be lost, reordered, and/or duplicated; and messages are not corrupted by the network (i.e. if a message is received, it has the same payload as specified at its origin). In our model, nodes are fail-stop.

In order to identify which nodes can communicate with one another, MojaveComm introduces an abstraction device called a *group* (see related work in Section 1.1). Conceptually, a group is a set of nodes. Using MojaveComm, two nodes may only communicate if they are both members of the same group simultaneously. The entire system may be comprised of many groups. It is possible for groups to have intersecting membership.

MojaveComm imposes a total order on all of the messages sent within a group. In this way, each node receives all the messages sent to the group in the same order (and without "gaps").

## 2.1 Groups & Views

Group membership is dynamic and nodes may join or leave a group at any time. Since the underlying communication mechanism is unordered and unreliable, from the perspective of individual nodes in a group, group membership membership is not always globally consistent. This local perception of group membership is known as a *view* (similar to views in [8], or subgroups in [9]).

Each node has exactly one view of each group of which that node is currently a member; a node is always in its own view. Moreover, a node only appears in a view of a group if that node was once a member of that group. To become a member of a particular group, a node simply includes itself as the only member of a singleton view corresponding to that group. It is possible for this initial

view to expand by merging this view with the views of other group members (this is described in Section 2.1.1).

In order to identify when two nodes "share" a view (i.e. when two nodes are members of the same view at the same time), we introduce an identifying tag, called an *epoch* number (or simply epoch), that we associate with a view so as to allow us to uniquely identify views with identical membership[1]. In this way, we can represent a view as a triple:

$$(G, M, e) \in \textit{group ID} \times \textit{node ID set} \times \textit{epoch}$$

For convenience we use the notation $G_e^M$ to denote the triple $(G, M, e)$. Note that two views, $G_e^M$ and ${G'}_{e'}^{M'}$, are equal iff each of their components is equal:

$$G_e^M = {G'}_{e'}^{M'} \equiv G = G' \wedge M = M' \wedge e = e'$$

Locally, each node keeps track of the set of views of which it is a member. For a given node $n$, we denote this set as $V(n)$; $\overline{V}(n)$ is the sequence of the values of $V(n)$ over time for node $n$. Also, since it is often necessary to talk about a particular node being a member of some view, we will introduce the operators $\in_V$ and $\in_{\overline{V}}$ to represent this fact (for current and past view membership, respectively):

$$n \in_V G_e^M \equiv G_e^M \in V(n) \wedge n \in M$$
$$n \in_{\overline{V}} G_e^M \equiv \exists V \in_{\langle\rangle} \overline{V}(n) : G_e^M \in V \wedge n \in M$$

Overall, there are two primary guarantees governing how view membership is related to group membership at a given instance in time (which follow directly from the statements above):

1. The intersection of all views of a group is a subset of the actual group membership.
2. The union of all views of a group is a subset of all the nodes that have ever been a member of that group (up to that point).

However, it is important to note that this definition allows for the possibility that a view of a group contains one or more nodes that are not currently members of the group. Such a view is known as a *stale* view. In general, it is not possible to detect accurately when a view is stale (see the discussion on failure detection in Section 2.1.2). For example, consider the scenario presented in Figure 2.1. Here, nodes $A$ and $B$ have split off from view $G_e^{\{A,B,C,D\}}$ to form views $G_{e+1}^{\{A,E\}}$ and $G_{e+1}^{\{B,F\}}$. Although the failure detection mechanism of MojaveComm will ensure that all remaining

[1]The epoch number basically serves as a logical clock (as presented in [14]).

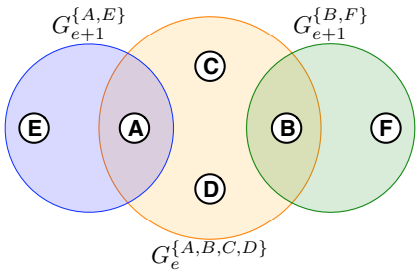



Figure 2.1: Multiple intersecting views

members of $G_e^{\{A,B,C,D\}}$, $C$ and $D$, will eventually move on to new views, there may be a period of time in which all three views depicted in the figure may exist simultaneously.

### 2.1.1 View Change Events

View membership may change for a variety of reasons: failure detection (i.e. when a process is suspected to be dead; see Section 2.1.2), an explicit view synchronization request (i.e. a group *Join* operation; see Section 2.3), or implicit synchronization initiated by remote instances of either of the former events. This type of view synchronization event (no matter how it is initiated) is known as a *view change*. When one of these synchronizations occurs, MojaveComm notifies the application of this change through a view change event.

Although the installation of a new view itself is ultimately a local operation, there are a number of global guarantees that MojaveComm enforces on a view change event. For example, consider the situation where a node $n$ receives a view change $G_{e'}^{M'}$. Furthermore, let us denote the previous view change $n$ has received as $G_e^M$ (where $e \neq e'$), and the sequence of messages $n$ has received since view change $G_e^M$ – but before the next view change – as $R_n$. In this case, MojaveComm guarantees the following property:

$$\begin{aligned} &\exists V, V' : G_e^M \in V \wedge G_{e'}^{M'} \in V' \wedge \langle V, V' \rangle \subseteq_{\langle\rangle} \overline{V}(n) \implies && \text{[VIRTUAL SYNCHRONY]} \\ &\quad \forall m \in M' : \exists V, V' : G_e^M \in V \wedge G_{e'}^{M'} \in V' \wedge \langle V, V' \rangle \subseteq_{\langle\rangle} \overline{V}(m) \implies R_n = R_m && (2.1) \end{aligned}$$

In other words, all nodes that shared a previous view over the course of a *single* view change are guaranteed to have received the same sequence of messages (i.e. the same messages in the same order) in the previous view[2].

Nevertheless, it is important to note that, when a node $n$ receives a view change event $G_e^M$, it

[2]Note that, in conjunction with the message delivery guarantees of Section 2.2, it is possible that failure to deliver messages in one group will result in a view change in another group (and may affect which nodes may be in the new view).

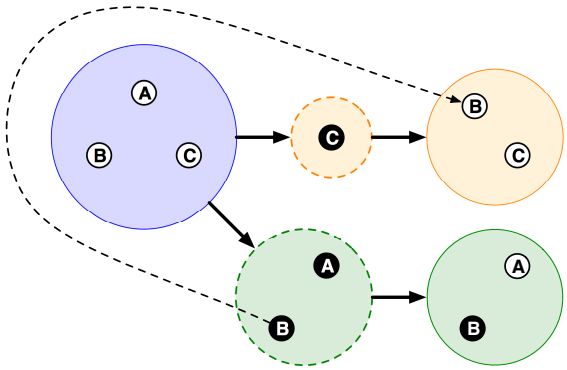



Figure 2.2: View synchronization that necessitates a *ghost* view change

is not guaranteed that every node $m \in M$ also has received such a view change. Rather, it is only guaranteed that it would be consistent if those nodes were to have received the same view change event $G_e^M$. We will refer to such a situation as a *ghost* view change.

Such ghost view changes are not directly observable by the application in the same way that normal view changes are (though, for a node $n$, they do appear in the view history $\overline{V}(n)$). Instead, they only serve to inhibit future unsafe synchronization in the event of failure; the application is never explicitly notified of such events. Thus, when a node $n$ receives a view change event $G_e^M$, MojaveComm ensures that every remote member $m \in M$ of the new view either has received a real or ghost view change event in $G$ corresponding to $G_e^M$.

Although we defer the discussion of the specifics of the implementation of the view management protocol (the protocol in the implementation that governs view synchronization and the corresponding view change events) to Section 3.2.3, it is worth introducing an example here to illustrate the necessity of these ghost view change events. Consider the following situation depicted in Figure 2.2. There are three nodes $A$, $B$, and $C$, that initially share a view. One node, $C$, splits off (initiating a view synchronization) and nodes $A$ and $B$ start to form a new view. $A$ and $B$ proceed to the stage of the protocol where they are prepared to commit the new view; however, they are still both waiting for a final *Commit* message (upon receipt of which they will actually commit the view). Node $A$ receives the *Commit* but $B$ doesn't. At this point $C$ has started, but not finished, its own view change and, by the time $B$ detects that it is missing a message from $A$, $B$ and $C$ continue a view synchronization together and both successfully commit the new view.

In the scenario depicted above, it is necessary for node $B$ to undergo a ghost view change with the membership $\{A, B\}$. Otherwise, without any intermediate view change, it is possible for node $B$ to receive a view change event with membership $\{B, C\}$ where the VIRTUAL SYNCHRONY property holds (according to the definition presented by Equation 2.1). However, it is not possible to guarantee that the VIRTUAL SYNCHRONY property can hold for two such views in general. This is because it is possible for nodes in the two separate view fragments to have received different subsets

of the messages that were sent in the previous view. In such a case, this would violate VIRTUAL SYNCHRONY; thus, we introduce the ghost view change mechanism to prevent this from occurring.

### 2.1.2 Failure Detection

As described in Section 2.1.1, failure detection is one of the triggers for a view change event. However, as a result of the underlying communication model that MojaveComm uses, it is not possible to perfectly determine when a node has actually failed. The fundamental problem, of course, is that messages may take arbitrarily long in transit from the source to the destination. Since there is no specific time limit within which all messages must be delivered, it becomes impossible to distinguish between a node that has crashed and one whose messages take a long time to arrive.

In such an environment, it is important that we are always able to detect a failed node in a finite amount of time; otherwise, it will be impossible to guarantee progress in general. Thus, we adopt an approximate failure detection mechanism whereby we are always able to identify nodes that actually have failed, but where we may inadvertently identify slow (but live) nodes as dead as well. Although such behavior is obviously undesirable, the incorrect detection of node failure does not affect the overall correctness of the guarantees that the MojaveComm protocol provides. Rather, it only degrades protocol performance and/or impedes application-level progress.

For example, if one node is connected to the rest of the nodes in the view membership over some slow and/or lossy physical link in the network, it is possible that the approximate failure detection mechanism will incorrectly identify this node as having failed. This, in turn, will trigger a view synchronization; during the view synchronization, transmission of pending application-level messages is suspended. In the best case, this reduces efficiency and slows down the message transmission rate. More severely, the slow node may even be ejected from the view, potentially halting any application-level progress on that node or others (in the event that the nodes require some specific view membership in order to advance). Overall, this situation only impacts the system liveness, not its safety.

## 2.2 Message Transmission & Delivery

Of course, the whole reason we care about view membership is because we would like to be able to make guarantees with respect to message ordering within a group. In particular, we would like to constrain the behavior of message transmission and delivery (to the application) in such a way as to ensure that the VIRTUAL SYNCHRONY property (see Equation 2.1) is met on each view change event received by the application.

In order to send a message, a node must first assign the message a position in the group total order. MojaveComm uses this sequence number as a mechanism by which to uniquely identify a

message and to constrain the ordering of message delivery. Thus, conceptually, a MojaveComm message can be represented as a triple: $(P, s, G_e^M) \in \textit{message payload} \times \textit{sequence number} \times \textit{view}$; for convenience, we denote this as $P_s : G_e^M$. In this way, the pair $(s, G_e^M)$ serves as a unique identifier for message $P_s : G_e^M$.

In addition to the virtual synchrony property, MojaveComm provides the following guarantees with respect to message transmission and delivery:

- For any two messages $P_i : G_e^M$ and ${P'}_j : G_e^M$ such that $i < j$, if a node $n$ receives both messages then $n$ will deliver $P_i : G_e^M$ before ${P'}_j : G_e^M$ (no matter the order in which they are actually received).
- For any two messages $P_i : G_e^M$ and ${P'}_j : {G'}_{e'}^{M'}$ that originate from node $n$ and are delivered to node $m$, if $i < j$ then $m$ will deliver $P_i : G_e^M$ to the application before it delivers ${P'}_j : {G'}_{e'}^{M'}$.
- For any two views $G_e^M$ and ${G'}_{e'}^{M'}$, if a node $n \in M \cap M'$ sends two messages $P_i : G_e^M$ and ${P'}_j : {G'}_{e'}^{M'}$ (in that order), then $i < j$.

In other words, there are two ordering constraints to which all communicating participants must adhere: the group order, and the node order. Thus, all nodes that share a view must receive all messages sent in that view in the same order; moreover, all nodes (regardless of view membership) must receive messages from the same node in the same order.

In order to enforce these ordering guarantees, MojaveComm adopts a policy that ensures the first two properties specified above are always met[3]. Consider the following scenario. Suppose that node $n$ is a member of groups $G_{i \in 0..m}$ and that $n$ has just received a message $P_j : (G_i)_e^M$. Node $n$ may only deliver $P_j : (G_i)_e^M$ under the following conditions:

- $P_j : (G_i)_e^M$ is the next message in the sequence for view $(G_i)_e^M$.
- It is known, for all views corresponding to groups $G_{k \in 0..m}$ $(k \neq i)$, that the next message to deliver in the sequence for view $(G_k)_{e'}^{M'}$, message ${P'}_s : (G_k)_{e'}^{M'}$, has sequence number $s > j$.

The policy above is sufficient to directly enforce the first two ordering conditions detailed above. The first component of the message transmission and delivery policy satisfies the group ordering condition. If two messages sent in a group are delivered out of order (such that a message with a higher sequence number is delivered before a message with a lower sequence number), then that delivery necessarily must violate the first policy component (since messages are assigned sequence numbers in strictly increasing order).

Moreover, the second component of the message transmission and delivery policy satisfies the node ordering condition. If a node delivers a message in one group then it is possible to prove that

[3]To ensure that the final property is met, MojaveComm simply assigns sequence numbers to outgoing messages in increasing order (though the sequence numbers need not be contiguous).

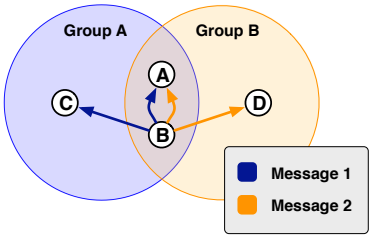



Figure 2.3: Node $B$ sends two messages in two different MojaveComm groups.

the message has the smallest sequence number of the next message to deliver in any other group of which that node is a member. Suppose that this most-recently delivered message is the message that was delivered out of order for a given node. Next, suppose that we receive another message in some group for that same node that was supposed to be delivered earlier. In that case, either the message was in the same group (in which case we already know from the argument above that this cannot happen) or the message is in a different group. In the case that the message was received in a different group, if this message was supposed to be delivered before our most-recently delivered message, then we know that it has a sequence number less than the one of that message. However, we will only deliver a message if we know that the message has a smaller sequence number than that of any other group of which that node is a member. As this leads to a contradiction, we know that this case is also not possible.

Consider the scenario depicted in Figure 2.3. In the diagram, there are two views: $A_e^{\{A,B,C\}}$ and $B_f^{\{A,B,D\}}$; node $B$ sends one message in each view. Without loss of generality, suppose $B$ sends message *1* before message *2*. In this case, we know that the sequence number for message *1* is strictly less than that for message *2*. Thus, MojaveComm guarantees that all nodes in the intersection of the two groups (i.e. $\{A, B\}$) receive the messages in the order $\langle 1, 2\rangle$. It is important to note, however, that MojaveComm does not constrain the exact timing of delivery, only the order. In this way, for example, it is possible for node $D$ to deliver message *2* before $A$ or $B$ even receives message *1*.

## 2.3 Application Interface

So far, we have described some of the message ordering guarantees that MojaveComm exposes to the application, but we have not presented any specifics with respect to the actual interface through which MojaveComm presents these features to the application. This section details an abstract specification of the MojaveComm application interface. Conceptually, the interaction of MojaveComm with the application can be grouped into two classes: application-initiated actions, and network-initiated events (notifications).

From an abstract perspective, MojaveComm supports the following actions:

- **Create** - Become a member of group $G$
  - *Input:* $Id_G$, the identifier representing group $G$, the group of which to become a member
  - *Output:* $H_G$, a unique handle representing the node's membership in group $G$
- **Destroy** - Cease to remain a member of group $G$
  - *Input:* $H_G$, a handle representing the node's membership in group $G$
- **Join** - Attempt to synchronize views (for group $G$) with nodes $N \cup M$ (where $G_e^M$ is the node's view of $G$)
  - *Input:* $H_G$, the handle to membership in group $G$; $N$, a set of identifiers representing nodes in group $G$ with which to attempt to synchronize views in addition to nodes in $M$
  - *Output:* $b$, a Boolean representing success or failure (i.e. $b \equiv \perp$ iff no view synchronization occurs)
- **Send** - Attempt to send a message $P$ to view $G_e^M$
  - *Input:* $H_G$, the handle to membership in group $G$; $P$, a payload message to deliver to nodes in $G_e^M$

and issues the following notifications:

- **Receive** - Receive a message $P$ in view $G_e^M$ from node $m \in M$
  - *Parameters:* $H_G$, the handle to membership in group $G$; $Id_n$, the identifier representing node $n$; and $P$, a payload message received from node $n$ in view $G_e^M$
- **View Change** - The local view $G_e^M$ of group $G$ has changed
  - *Parameters:* $G_e^M$, the new local view of the membership of group $G$; $M_{vs} \subseteq M$, the set of nodes for which the antecedent of the VIRTUAL SYNCHRONY property holds; $i$, an optional message identifier representing the last message sent in the previous view (if known, otherwise $i$ is *None*)

# Chapter 3

# MojaveComm Implementation

This chapter provides design and implementation details for the reference implementation of MojaveComm (available via subversion at [2]). The code-base for the reference implementation is written in OCaml [3], and can be compiled to run on a variety of different platforms; for the time-being, however, only Unix-based systems are supported.

The implementation is designed to adhere to the specification presented in Chapter 2. Of course, as such an abstract description is not suitable to describe an actual working model of the protocol, Chapter 3 describes in detail the design decisions we have made in our MojaveComm implementation.

## 3.1 Design Choices & Goals

Although the primary goals of the MojaveComm protocol are embodied in the guarantees detailed in the abstract specification presented in Chapter 2, there are a number of secondary considerations that we would like to take into account during the design of an actual realization of the protocol. In particular, we would like to ensure that the implementation of MojaveComm is fully distributed (i.e. there is no central point of failure), modular and configurable, and implemented on top of the existing network infrastructure. These objectives, although ultimately orthogonal to the message delivery guarantees of Chapter 2, are nevertheless important aspects of the design of the protocol and serve to improve the reliability, scalability, performance, and ease of adoption of the MojaveComm implementation.

One of the primary design considerations is the choice to utilize IP-multicast as the underlying network transport mechanism for low-level MojaveComm message passing. Since MojaveComm does not make strong assumptions about the underlying communication model in order to provide its guarantees, many of the existing network protocols would be sufficient to use in the MojaveComm implementation. In the interest of efficiency, it would be ideal to choose a protocol that provides the most guarantees of which MojaveComm can take advantage, while adding the least amount of overhead. The IP-multicast protocol appears to be a natural fit for this role, since MojaveComm

can clearly take advantage of the IP-multicast broadcast communication facility (as messages need to be delivered to all members of each view) – and since IP-multicast provides exactly the best-effort message delivery required by the MojaveComm abstract specification.

Another driving influence in the design of the protocol is the desire to ensure that MojaveComm is fully distributed. In this way, we would like to avoid the use of any specific coordination infrastructure that would introduce a central point of failure; such bottlenecks generally serve to decrease reliability and scalability as a coordinator may fail or become overloaded. Moreover, a fully distributed solution does not require any additional configuration to identify nodes with "elevated" privileges and/or responsibilities in the network in order to function properly (although we still need to solve the bootstrapping problem; see Section 3.2.3).

Given that the abstract MojaveComm specification does not tightly constrain many specific design choices (such as those mentioned above), it is important to make the implementation of MojaveComm modular and configurable. In this way, it is possible to easily expand the MojaveComm implementation and to customize the behavior of the protocol as a whole in order to fit the needs of a specific MojaveComm deployment. For example, although we strive in the implementation we present in this text to avoid any centralized coordination, it is possible that a specific deployment environment has a set of ultra-reliable high performance nodes that would make communication more efficient if MojaveComm were able to exploit these. In such a case, the we intend that the MojaveComm implementation architecture is flexible enough to accommodate this.

## 3.2 Layers

Conceptually, MojaveComm is organized into the following set of components: the transport, sequencer, view manager, and application modules. Each of these components presents an abstract interface with which the other components interact. Each module has the following responsibilities:

- **Transport** - The transport module provides a reliable multicast primitive.
- **Sequencer** - The sequencer module handles the sequencing and reordering of all incoming and outgoing messages; it also keeps a cache of received messages (both delivered and undelivered).
- **View Manager** - The view manager keeps track of the view membership and coordinates view changes.
- **Application** - The user provides the application module to drive the operation of MojaveComm and to react to any notifications MojaveComm generates.

In Figure 3.1, we present a block diagram of the modules detailing the interactions between them. The *Sequencer* and the *View Manager* comprise the topmost layer of MojaveComm. These

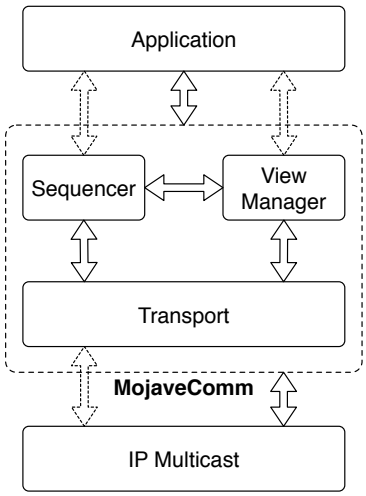



Figure 3.1: The layered architecture of MojaveComm.

two components interact directly with one another and each expose a fraction of their interface to the *Application* layer (though this is fully transparent as there is only one combined interface exposed to the user). The *Sequencer* and the *View Manager* both communicate with the *Transport* layer in order to send and receive both component-specific and application payload messages. Ultimately, the *Transport* layer rests on top of standard *IP Multicast*; the *IP Multicast* layer is responsible for the actual low-level message exchange. The dotted line in the figure that circumscribes the *Sequencer*, *View Manager*, and *Transport* layers indicates the interface boundary with which external components interact; in the implementation, there exists some glue code which presents a unified interface to the external environment.

In the remaining sections of this chapter, the text details both the interface that each of the modules exposes, as well as the specifics of the actual protocol implementation for each of the components of the MojaveComm reference implementation.

### 3.2.1 Transport

The transport module is the bottommost layer of the MojaveComm protocol. This layer is responsible for providing a reliable point-to-multipoint communication mechanism. Reliability, in this context, refers to two features: failure detection/notification, and automated message retransmission. The interface for the transport module supports the following actions:

- **Select** - Wait for an I/O event (*read/write*) on *tdesc*, the specified transport descriptor.
- **Send** - Send an outgoing message, $P$, addressed to some set of nodes, $M$, on the specified transport descriptor, *tdesc*.

and issues the following notifications:

- **MessageReceived** - Receive a message, $P$ from node $n$ on transport descriptor *tdesc*
- **XmitFeedback** - Receive notification of the set of nodes, $M$, for for which it is not possible to guarantee that the message $P$ was successfully delivered to such nodes.
- **SelectEvent** - Receive notification that descriptor *tdesc* has pending I/O events (i.e. *read/write*).

#### 3.2.1.1 MojaveComm Reliable Multicast Protocol

The implementation of the transport protocol for the MojaveComm reference implementation is called the *MojaveComm Reliable Multicast Protocol* (or MCRMP). MCRMP is a negative acknowledgment (NACK) based point-to-multipoint communication protocol with congestion control implemented on top of IP-multicast (modeled after the protocols in [11] and [13]). MCRMP provides one primary guarantee: if a message $P$ is sent from one node $n$ to a set of nodes $M$, then *eventually* $n$ will receive feedback in the form of a set $F = \{f | f \in M \wedge n$ was not able to prove $f$ received message $P\}$. Note that this feedback mechanism is consistent with the approximate failure detector of Section 2.1.2.

MCRMP has four basic types of messages: payload, NACK, NACK-reply, and heartbeat. *Payload* messages are responsible for transporting upper-layer message payloads to other nodes in the system. MCRMP uses *NACK* messages to let a sender know that a recipient has not received a given payload message; and MCRMP uses *NACK-reply* messages to retransmit NACKed messages. Each node using MCRMP periodically sends *heartbeat* messages to all other nodes with which it is actively communicating for the purpose of failure detection and to update its retransmission cache.

Conceptually, the MCRMP message types have the following format:

- **Payload**(*ID*, *source*, *destination set*, *data*)
  - *ID* - An identifier uniquely identifying this message with respect to the source
  - *source* - The node identifier representing the origin of the message
  - *destination set* - A set of node identifiers, representing the destination(s) of the message
  - *data* - The message payload
- **NACK**(*NACK ID*, *source*)
  - *NACK ID* - The identifier of the message to negatively acknowledge
  - *source* - The origin of message to negatively acknowledge
- **NACK-reply**(*NACK ID*, *source*, *data*)
  - *NACK ID* - The identifier of the message to re-send
  - *source* - The origin of the message to re-send

  - *data* - The payload of the message to resend

- **Heartbeat**(*source*, *delivered ID*)
  - *source* - The origin of the message
  - *delivered ID* - The identifier of the last message delivered at *source* (for which the origin was the recipient of this message)

#### 3.2.1.2 Advanced Features

The description of the MCRMP message types presented in Section 3.2.1 is somewhat of an oversimplification. In particular, MCRMP supports both message fragmentation and congestion control – two features that require additional communication beyond that which has already been described. Although these features do not directly impact the guarantees provided by MojaveComm, they play an instrumental role in ensuring high performance and usability for the protocol.

Message *fragmentation* allows MCRMP to split up a message into many smaller pieces (or fragments) in order to send them in multiple distinct IP-multicast packets. One of the practical considerations of introducing a message fragmentation mechanism at this layer is that IP-multicast has constraints on the maximum size of a multicast message [18]. Thus, since MCRMP uses IP-multicast to send all of its low-level messages, if the size of a MCRMP message exceeds this limit then that message will not be delivered. In order to prevent such a situation from occurring, MCRMP allows the *destination set* and the *data* fields of a payload message to be split up into multiple smaller chunks and exchanged via separate packets over the network; MCRMP recombines these fragments on the recipient end.

Also, MCRMP makes use of a rate-based congestion control algorithm in order to ensure that no message recipient is being flooded with more messages than it can handle. In the event that MCRMP detects this scenario (by paying attention to message transmission failures), it will limit the transmission rate of the sender in order to compensate. To ensure that the rate does not get stuck in a local minimum, MCRMP constantly performs an AIMD (additive increase, multiplicative decrease) search of the available bandwidth.

### 3.2.2 Sequencer

The sequencer module is one of the two middle-layer components of the MojaveComm protocol stack (the other is the view manager module; see Section 3.2.3). The sequencer is primarily responsible for ordering payload messages that are sent using the MojaveComm protocol. For outgoing messages, this involves assigning such messages appropriate sequence numbers; for incoming messages, the sequencer is responsible for reordering the messages to ensure that messages are delivered in the right sequence to the application.

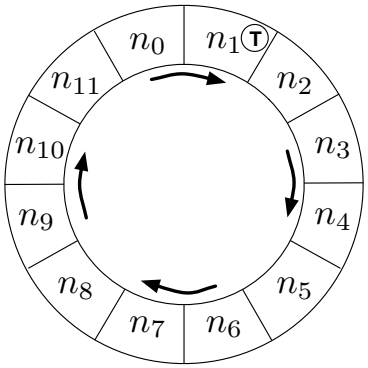



Figure 3.2: A twelve-node view arranged into a ring by the token sequencer protocol.

As one of the responsibilities of the sequencer is to reorder incoming messages for in-order delivery to the application, the sequencer must also perform some message caching (to postpone the delivery of out-of-order messages). Since the sequencer is required to do this caching anyway to properly reorder messages, this cache serves a dual purpose and is consulted by the view manager module during the message synchronization phase of the view synchronization protocol (see Section 3.2.3).

In order to assign sequence numbers to outgoing messages, the sequencer module relies on a token-based mutual exclusion protocol. The basic functionality of the protocol behaves as follows. Associated with each unique view is a token. The protocol guarantees that at most one node in each view has possession of (or "owns") the token at one time. Upon a view change, one node from the view membership is selected to initially take possession of the token for that view. The actual protocol uses the fact that node identifiers are unique and totally ordered; so, by default, the protocol chooses the node with the smallest identifier in the view as the initial owner of the token. The protocol then arranges the view into a ring, where the successor of node $n$ is the node with the next-largest identifier to $n$ (or the node with the smallest identifier, if $n$ has the largest identifier in the view). Figure 3.2 depicts the formation of such a ring for a view of twelve processes; $n_1$ holds the token.

Associated with the token is a pair of values that travels around with the token as it is passed from one node to another in the ring. One value represents the last sequence number assigned to a message in this view (or a special value, *None*, if no messages have been assigned a sequence number). The other value is the first available (i.e. unused) sequence number that may be assigned to a message.

From the moment that a node assumes ownership of the token, it may assign sequence numbers to pending outgoing messages and update the two token values appropriately. In order to determine the exact placement of a message within the sequence of messages for a particular view, MojaveComm includes both the sequence number of the message and the sequence number of the previously

sequenced message (in that view) in the message header. To ensure that each node in the view gets a chance to sequence its messages, a node is only allowed to possess the token for at most some maximum amount of time (this is configured statically; the value may be a function of the view size). When a node relinquishes ownership of the token, it passes the token (along with its associated satellite data) to the next node in the ring of nodes that the view has formed. It is possible that communication failure may occur in attempting to pass the token on to the next node. In this case, the nodes will detect the transmission failure, initiate a new view synchronization (see Section 3.2.3), and create a new token for the next view.

Of course, assigning sequence numbers to outgoing messages is only one of the responsibilities of the sequencer. The other primary function that the sequencer must perform is to reorder incoming messages for in-order delivery to the application. According to the abstract specification of the message delivery guarantees presented in Section 2.2, message delivery is subject to two ordering constraints: node order, and view order.

In order to adhere to these constraints, the sequencer adopts the following policy. When a node receives the token in some view, it increases the next available message ID to be at least as large as one greater than the largest sequence number of any message the node has currently received (in *any* view, regardless of whether or not this message has been delivered) so far (unless the current value is already at least this large); while the node has possession of the token, this value is also updated similarly on receipt of any subsequent messages (again, in any view). In this way, it is possible for the node to locally establish a lower bound on the sequence number of the next message to be sequenced in each view.

The sequencer uses this lower bound on the sequence number of the next message in each view to determine if the second ordering constraint of Section 2.2 is satisfied for a given message that is a candidate for delivery. It is possible for the node to determine if the first constraint is satisfied directly, as each message contains both the sequence number for that message and the sequence number for the previous message sent in that view.

In practice, the sequencer does not have an infinite amount of space to use as a cache for incoming messages to reorder. Unfortunately, this means that it is possible that the sequencer will have to drop some messages. The policy we choose in the implementation is to first drop any delivered messages from the cache (in ascending order). Then we drop undelivered messages (in descending order). This ensures that we always make room for undelivered messages (in the hopes of delivering them) and that we drop undelivered messages with the least chance of being delivered (since we have to deliver those messages before it, at least). Discarding an undelivered message ultimately results in a new view synchronization.

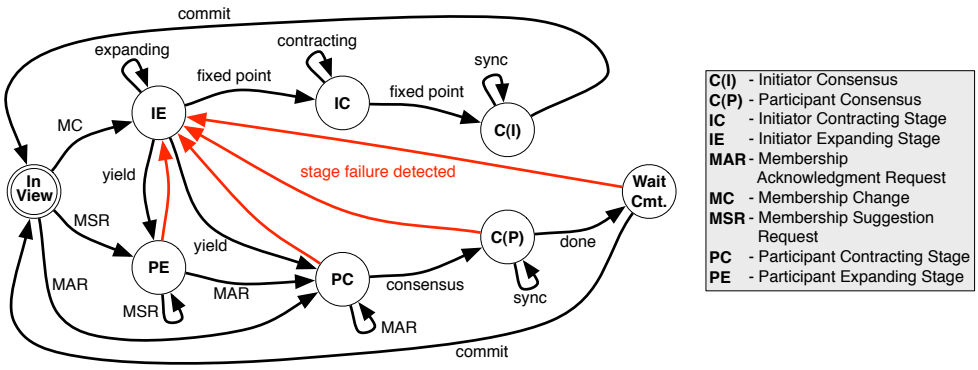



Figure 3.3: This figure represents the state machine for the view management protocol.

### 3.2.3 View Manager

The view manager is the component of MojaveComm that is responsible for performing the view synchronization required by the abstract protocol specification detailed in Chapter 2. Although the view manager is primarily responsible for arbitrating consensus on view membership when nodes fail or new nodes join, the view manager is also instrumental in enforcing the VIRTUAL SYNCHRONY guarantee (see Equation 2.1). This involves both calculating a new view membership, identifying the subset of nodes for which the VIRTUAL SYNCHRONY property will hold, and then making sure that these nodes have the same set of messages sent in the previous view.

The view manager implementation embodies this process in a series of four stages: *Expanding*, *Contracting*, *Consensus*, and *Wait Commit*. Figure 3.3 depicts the state machine for the protocol. Conceptually, there exists two "paths" from the beginning to the end of a view synchronization using this protocol. One path is for the *initiator* of a view synchronization; the other is for a view synchronization *participant*. Any node may be either an initiator or a participant. And, although a node is never both an initiator and a participant (at least within the context of a single group) at the same time, a node may assume both of the roles over the course of a single view synchronization.

Initially each node starts out in the *In View* state, where the node is not actively performing any view synchronization. However, when a node is in this state, one of two events may trigger a view synchronization: either one node issues a *join request*, or a node failure is detected. Join request messages serve to notify members of one view that one or more members of another view wish to merge views[1]. If a process is detected to have failed, then one or more members of the view initiate a view change and try to exclude the process from the view.

If a node is the one to initiate a view synchronization, it assumes the role of the *initiator*, and takes the upper path depicted in Figure 3.3. The initiator is responsible for coordinating the view synchronization. Multiple nodes may simultaneously attempt to initiate a view change. In this event, the view management protocol has a mechanism that allows one initiator to *yield* to another

[1]Note that there is no corresponding operation to split views; we do not permit views to fragment at will.

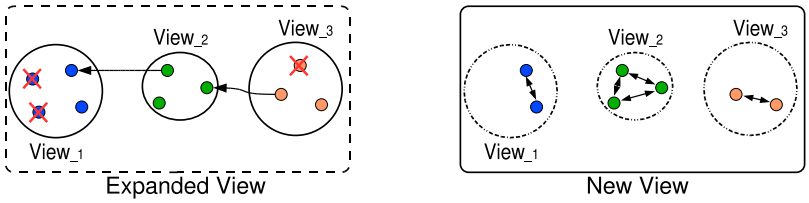



Figure 3.4: Multiple views merging

so that a single initiator is elected. Unfortunately, because of the possibility of transient failures, it is impossible to guarantee that the initiator election will result in exactly one initiator; in this case, multiple view synchronizations proceed simultaneously.

Nevertheless, one major advantage of our protocol is that it allows changes in view membership to include, at the same time, multiple processes joining and leaving the view. For example, Figure 3.4 illustrates how, in one step, three views merge and some nodes from each view will be excluded as they fail during the view change process.

### 3.2.4 Protocol Stages

The purpose of the first stage, the *Expanding* stage, of the view synchronization process is to collect suggestions from the current members of the view and, from these suggestions, ascertain what the new membership of the view should be. This stage is repeated until a fixed point is reached (nodes are only added to the membership with each round of suggestions). In the example presented in Figure 3.4 this is shown in the drawing to the left. At the end of the expanding stage, the expanded view contains all the members of the three initial views.

During the *Contracting* stage, processes that have failed or that want to leave the group are removed from the maximal fixed point view reached during the previous stage. The goal of this stage is to reduce the membership of the view to the current set of active processes. It is important to note that between a process expressing interest in joining a group and the commit of the view change, that process could fail or there might be a network partition – in which case more than one process might need to be excluded from the view. In Figure 3.4 the *new view* illustrates the membership after the contracting stage, where failed nodes have been evicted from the final view.

The *Consensus* stage is critical for preserving VIRTUAL SYNCHRONY. During this stage, processes that have survived so far agree on what messages they need to deliver to the application layer to guarantee that all members of the view that survive the view change have delivered the same set of messages in the previous view. The consensus stage is illustrated by the arrows in the second drawing of Figure 3.4; here, each surviving process synchronizes with all the other processes in its previous view.

In the last stage, the new view to be installed is broadcast to all of its members and is locally

installed by each member. The view change initiator sets the epoch of the new view to be larger than the largest epoch involved in the view change. Also, the sequence number is reset to 0 and the view manager propagates a view change event to the upper layer with the new epoch and the new sequence number is broadcast to all members of the new view. Upon receiving the view change event each node delivers it to the application (the upper layer running on top of MojaveComm).

# Chapter 4

# MojaveFS: A Case Study

MojaveFS [21] is a network-based distributed filesystem designed with the intent to be highly flexible and fault tolerant while retaining the familiar read/write semantics of traditional local, centralized filesystems. In particular, MojaveFS aims to provide location transparency and automatic data replication; replicas may be added to or removed from the system dynamically without need to stop the filesystem in order to perform the reconfiguration. Moreover, the ordering of all file operations are bound by a strict consistency model that makes MojaveFS a drop-in replacement for traditional storage solutions.

Because of its distributed nature and strict ordering guarantees, we consider MojaveFS to be a good candidate for the type of application that would most be able to take advantage of the MojaveComm protocol and the communication guarantees that it provides. In this chapter, we present a high-level overview of the design and implementation of the MojaveFS filesystem, with particular emphasis on the role that MojaveComm plays with respect to facilitating the file-access semantics of MojaveFS. It is important to note that this chapter is not intended to provide an exhaustive description of MojaveFS and/or its implementation; rather, this text aims to expose enough details so as to give the reader sufficient context to understand how MojaveComm is used in this system.

## 4.1 Overview

In most respects, MojaveFS acts like a typical Unix filesystem. For example, MojaveFS supports the usual filesystem abstractions like files, directories, access permissions, and metadata. In terms of the actual implementation, MojaveFS is written as a FUSE [1] module[1] and, when mounted, is presented as a traditional Unix-style mount point. Thus, to access files stored in MojaveFS, a user simply needs to mount an instance of MojaveFS and access files in the usual manner via the regular

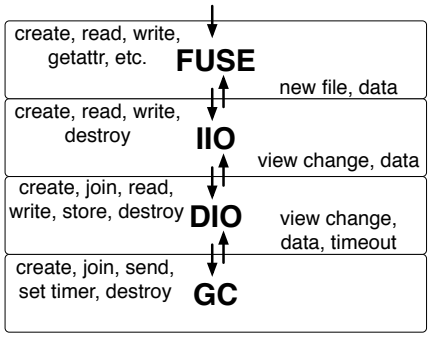



Figure 4.1: The layered architecture of MojaveFS.

system calls for file manipulation.

The major differences between MojaveFS and most traditional local filesystems is that the file data, file metadata, and the directory hierarchy itself is stored in a distributed manner across multiple physical machines. In fact, this filesystem state may even be replicated in multiple places. In this way, although this data distribution and replication is fully transparent from the user's perspective, from the perspective of the administrator MojaveFS provides a rich environment for tailoring the data distribution and replication policy necessary for a specific deployment environment.

Conceptually, each file and directory in a MojaveFS filesystem is served by a unique *virtual server*[2]. A virtual server is comprised of multiple physical nodes; these nodes all serve as consistent replicas of the file/directory. When a node contributes to the operation of a virtual server in this way, it is said to be a *member* of that virtual server. A node may dynamically join or leave the membership of a virtual server at any time. Moreover, a node may be a member of multiple virtual servers simultaneously.

Virtual servers serve as the infrastructure for MojaveFS; they provide the storage back-end. However, nodes that mount the filesystem and access the files that the virtual servers provide are known as clients. In practice, a member of a virtual server may also be a client.

In order to present the illusion that MojaveFS acts like a traditional local filesystem, MojaveFS adopts a strict consistency model for the ordering of all file operations. Specifically, MojaveFS conforms to a consistency model called *sequential consistency* (see Section 4.2 for a detailed description). Consistency is maintained at two levels: within each virtual server, and between virtual servers (this is important when a client is using multiple virtual servers simultaneously).

From an implementation point of view, MojaveFS is split into three major components: naming/lookup, replication/data consistency, and operation-sequencing. Figure 4.1 depicts the layered

[1]The FUSE project enables developers to write filesystems that execute their code outside of the operating system kernel. FUSE implementations are currently available for both Linux and Mac OS X.

[2]Actually, files are split into fixed-sized chunks, each of which is served by a separate virtual server; this is done for performance reasons. For the purposes of this text, however, we pretend there is only one virtual server per file. The distinction between chunks and files only serves to unnecessarily complicate the discussion.

architecture of the MojaveFS implementation; the bottom three layers correspond roughly to the three components listed above, respectively. In particular, the *Direct I/O* (or DIO) layer is responsible for maintaining consistent copies of replicated data and, in conjunction with the *Group Communication* (or GC) layer, enforcing the overall operation-sequencing guarantees of MojaveFS.

## 4.2 Sequential Consistency

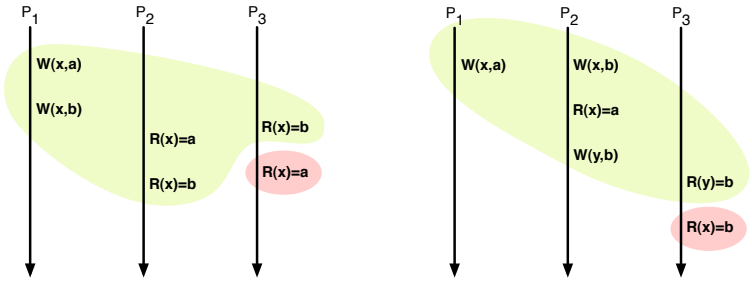
It is often the case that distributed/concurrent systems are more difficult to reason about than their centralized, sequential counterparts. For the most part, this difficulty stems from the unnatural evaluation semantics of such systems. In general, it is possible for operations of a node to be observed in a different order on different nodes. In order to guarantee some agreement on the order of operations, it is necessary for all the nodes in the system to adhere to an agreed-upon consistency model.

The more *strict* the consistency model is, the more restrictions it imposes (and the more guarantees it provides) on the ordering of operations each node performs with respect to one another. Generally speaking, in a distributed system, the choice of consistency model impacts the efficiency of global operations within the system (because it may be necessary to postpone some operations, as their execution may violate the consistency policy); the more strict the model, the less efficient the system may be.

Nevertheless, in spite of the potential for reduced efficiency, it is generally easier to reason about a system that adheres to a strict consistency model. In particular, the "gold standard" consistency model in a distributed system is a model that closely resembles that of a centralized system. In the *strict consistency* model, operations are required to be executed in the exact order that they are issued on the remote nodes (i.e. the order the operations would be issued on a traditional centralized system). However, the primary disadvantage of strict consistency is that it is only possible to be implemented in a distributed environment where there is a perfectly synchronized globally-shared clock available. Unfortunately, this is not typically available to most distributed systems and, worse, not even possible to achieve over unreliable communication channels (i.e. those where messages may be dropped) [10].

However, there is another consistency model, *sequential consistency*, that is more acceptable from an implementation-perspective for distributed systems which operate in an environment without a global synchronized clock. Consider the situation where a distributed system is comprised of a set of $N$ nodes (each of which has a local program from which it issues a set of operations). In this case, sequential consistency guarantees the following two ordering constraints [15]:

1. For all nodes $n \in N$, if $n$ issues operations $\langle o_1..o_k \rangle$ then all nodes $m \in N$ will observe $\langle o_1..o_k \rangle$ in order.




(a) Violation 1 (b) Violation 2

Figure 4.2: Violating the sequential consistency constraints

2. There exists a global sequence of operations $\langle o_1..o_k \rangle$ such that, for every node $n \in N$, $\langle o_1..o_k \rangle$ is consistent with the observations of $n$.

In other words, all nodes see the operations of every other node in the order in which a node issues those operations. Moreover, there exists some global order of operations that is consistent with what each of the nodes individually observes.

Consider Figures 4.2(a) and 4.2(b); each represents a distributed system with a sequentially inconsistent global execution. The operation highlighted in red indicates an operation that, if removed, would restore sequential consistency to the system. Specifically, Figure 4.2(a) violates property 1 above and Figure 4.2(b) violates property 2.

### 4.2.1 Ensuring Consistency in MojaveFS

MojaveFS makes use of two mechanisms to ensure that operations are processed in a sequentially consistent order. The first is that MojaveFS maps each virtual server to a MojaveComm group; file operations are sent as MojaveComm messages and, thus, are subject to the MojaveComm message-ordering constraints. The second is that MojaveFS adopts an *authority policy* to guarantee that at most one view of a group, the most up-to-date view, is able to make progress at any given time (i.e. the view that is authoritative)[3]. The paper that first describes this approach is [22].

The first mechanism alone actually does most of the work for MojaveFS with respect to providing the sequential consistency guarantees. This is evident from the message-ordering guarantees of MojaveComm presented in Section 2.2. For example, MojaveComm makes sure that all messages sent from a single node are observed in the same order by all nodes (both within and between groups). In this way, provided that MojaveFS issues MojaveComm messages for corresponding file operations in the same order that those operations are requested, the first requirement for sequential consistency is satisfied.

[3]Note that the authority policy is a special type of *primary* view (see [20]) where we base our decision on both majority membership and possession of the most recent message sent in the view.

However, it is important to note that MojaveComm alone does not satisfy the second requirement for sequential consistency. The main obstacle is that MojaveComm allows for there to exist multiple views of the same group. If these views correspond to a single virtual server and if they are allowed to make progress (i.e. process file operations), then it becomes impossible to come up with a single global sequence of operations that is consistent with the observations of all the nodes in the system.

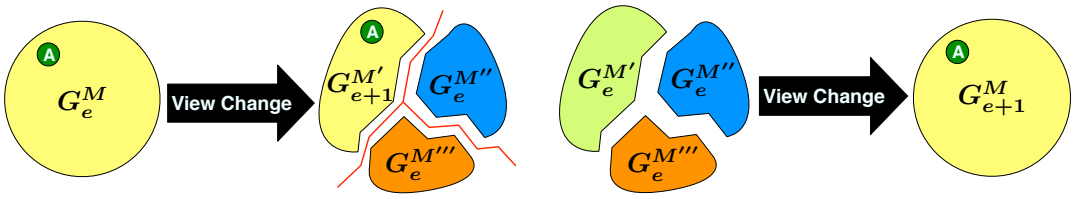
In order to prevent such a situation from occurring, MojaveFS makes use of the information that MojaveComm passes upwards on a view change event. Specifically, MojaveFS adopts the following policy enforced by the *DIO* layer of MojaveFS. When a new file is created, the initial view of the corresponding MojaveComm group is marked as authoritative (each process keeps track of this status locally). When a node receives a view change, MojaveFS considers the new view to be authoritative when:

- There exists a node in the view that has received the most recent message sent in the previous authoritative view.
- The view contains a majority[4] of the nodes from the previous authoritative view.

When a node receives a view change event from MojaveComm, it must gather information from other nodes in order to determine if the above authority conditions are satisfied. The protocol for determining view authority is split into three stages (effectively resulting in a two phase commit). In order to facilitate this, each node $n$ keeps track of a few attributes locally (for each group of which it is a participant): the last authoritative view ($A_e^M$) $n$ participated in (if one exists), the message ID of the last message $n$ received in $A_e^M$, and the message ID of the last message sent in $A_e^M$ (if known).

In the first stage, each node $n$ contributes its view, $A_e^M$, and the message ID of the last message sent in $A_e^M$ (or *None*, if $n$ does not have this information; see Section 2.3) to the other members of the new candidate authoritative view. After a node receives the contributions from all the other nodes in the view, it can make a local determination regarding whether or not the new view can be marked as authoritative. In the second stage, each node contributes this local determination on the authority of the new view; additionally, if at least one of these nodes considers the new view to be authoritative, one of those nodes also contributes the latest copy of the data associated with the virtual server corresponding to this group. Once a node has received all of the messages from the second stage, each node knows whether or not the new view is authoritative. In the third stage, each node acknowledges the receipt of all of the messages from the second stage. When all nodes receive all of the messages from stage three, the authority of the new view is committed; if the view is marked as authoritative, then the nodes in the view can process new operations at this point.

[4]For our definition of *majority*, we allow the view to have half of the previous nodes, provided it also contains the node with the smallest identifier (remember that node identifiers are totally-ordered).




(a) A three-way split of a virtual server group due to a network failure.

(b) Three non-authoritative views merge to create a new authoritative view.

Figure 4.3: A MojaveFS virtual server undergoes splitting & merging, altering its authority status (indicated by the "A" circumscribed by a circle).

In order to ensure that at most one view can be marked as authoritative at a given time, as soon as a node sends a message as part of the third stage in this protocol it moves into a *pre-commit* state where the node effectively forgets about the previous authoritative view of which it was a member. This is necessary to prevent a situation where some nodes commit the view and others do not. In such a case, if the nodes that do not commit are permitted to form a new authoritative view, then it is possible for two authoritative views for the same group to exist at the same time.

It is important to note that it is possible to end up in a state where all views lose their authority status and it is not possible to automatically reconstitute an authoritative view from the non-authoritative fragments.

# Chapter 5

# Conclusion

Group communication facilities, in general, provide strong message-ordering guarantees that may serve as the basis for the implementation of other applications and/or services. In this regard, MojaveComm is no exception. In particular, as we can see from the discussion in Chapter 4 (specifically, Section 4.2), MojaveComm provides sufficient guarantees to ensure that MojaveFS only needs to introduce minimal effort on its own part in order to provide guarantees above and beyond the native MojaveComm services.

Specifically, the message-ordering guarantees of MojaveComm lend themselves directly to the enforcement of sequential consistency in MojaveFS. In the event that the group membership is static and there are no node failures, MojaveComm does most of the work with respect to ensuring sequential consistency among the MojaveFS files. Rather, it is in the implementation of the MojaveFS "authority" policy where MojaveFS needs to do some extra work to ensure that recovery does not introduce extra views of a virtual server group that are able to make progress (thereby violating sequential consistency).

It is here that the flexibility in how MojaveComm allows for multiple views of a single group seems to complicate the MojaveFS design. The lack of direct support for a *primary view* abstraction puts the onus on MojaveFS itself to provide this feature. Although this makes the implementation of services like MojaveFS more cumbersome, we made a specific design decision not to include support for primary views directly in MojaveComm. One of the motivating factors is that the typical definition of a primary view based simply on maintaining a majority of the available processes in a group is not sufficient for ensuring sequential consistency in MojaveFS.

For example, consider a two-way split of a view corresponding to a MojaveFS virtual server. And suppose that the smaller view fragment of has delivered more messages than the larger one, where at least one of those messages corresponds to a *write* operation. Then, if the larger view fragment continues to make progress after the split, and if the first operation is a *read*, this is a situation that violates sequential consistency (since the members of the larger fragment did not see the write that happened in the smaller fragment).

Of course, it is not possible to know what sort of *primary view* policy a given application might require in general. Moreover, even if this were not the case, it may be unreasonable to impose the extra overhead of maintaining a primary view policy on applications that do not need this feature. Thus, in the design of MojaveComm, we opted simply to provide enough information in the view change event notifications to allow the application to keep track of its own notion of a primary view, if desired.

Overall, it seems as though the message-ordering guarantees of MojaveComm are certainly useful in the development of distributed applications. However, the lack of direct support for elevating the status of some view(s) of a group proves to be a hindrance. In future work, it might be worth exploring the use of a mechanism for specifying a policy for governing primary view selection in a group. Such a feature would likely simplify the implementation of distributed applications such as MojaveFS.

# Bibliography

[1] FUSE: Filesystem in userspace. http://fuse.sourceforge.net/. 4.1

[2] The MojaveComm reference implementation. http://mojave.caltech.edu/svnroot/mojave/mojavecomm. 3

[3] The OCaml programming language. http://www.ocaml.org. 3

[4] Yair Amir, Claudiu Danilov, and Jonathan Robert Stanton. A low latency, loss tolerant architecture and protocol for wide area group communication. In *DSN '00: Proceedings of the 2000 International Conference on Dependable Systems and Networks (formerly FTCS-30 and DCCA-8)*, pages 327–336, Washington, DC, USA, 2000. IEEE Computer Society. 1.1, 1.2

[5] Kenneth P. Birman. Isis: A system for fault-tolerant distributed computing. 1986. 1.1

[6] Kenneth P. Birman, Robert Constable, Mark Hayden, Jason Hickey, Christoph Kreitz, Robbert Van Renesse, Ohad Rodeh, and Werner Vogels. The Horus and Ensemble projects: Accomplishments and limitations. Technical report, Ithaca, NY, USA, 1999. 1.1, 1.2

[7] David R. Cheriton and Willy Zwaenepoel. Distributed process groups in the V Kernel. *ACM Trans. Comput. Syst.*, 3(2):77–107, 1985. 1.1

[8] G. V. Chockler, N. Huleihel, and D. Dolev. An adaptive totally ordered multicast protocol that tolerates partitions. In *PODC '98: Proceedings of the seventeenth annual ACM symposium on Principles of distributed computing*, pages 237–246, New York, NY, USA, 1998. ACM. 1.1, 1.2, 2.1

[9] Paul D. Ezhilchelvan, Raimundo A. Macedo, and Santosh K. Shrivastava. Newtop: A fault-tolerant group communication protocol. In *International Conference on Distributed Computing Systems*, pages 296–306, 1995. 1.1, 2.1

[10] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985. 1, 4.2

[11] Sally Floyd, Van Jacobson, Ching-Gung Liu, Steven McCanne, and Lixia Zhang. A reliable multicast framework for light-weight sessions and application level framing. *IEEE/ACM Transactions on Networking*, 5(6):784–803, December 1997. 3.2.1.1

[12] Hector Garcia-Molina and Annemarie Spauster. Ordered and reliable multicast communication. *ACM Transactions on Computer Systems*, 9(3):242–271, August 1991. 1.1

[13] Van Jacobson. Congestion avoidance and control. In *ACM SIGCOMM '88*, pages 314–329, Stanford, CA, August 1988. 3.2.1.1

[14] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978. 1

[15] Leslie Lamport. How to make a multiprocessor that correctly executes multiprocess programs. *IEEE Trans. Comput. C*, 28(9):690–691, 1979. 4.2

[16] Shivakant Mishra, Larry L. Peterson, and Richard D. Schlichting. Consul: A communication substrate for fault-tolerant distributed programs. Technical Report TR 91-32, Tucson, AZ (USA), 1991. 1.1

[17] Krzysztof Ostrowski and Kenneth P. Birman. Scalable group communication system for scalable trust. In *STC '06: Proceedings of the first ACM workshop on Scalable trusted computing*, pages 3–6, New York, NY, USA, 2006. ACM. 1.1

[18] Jon Postel. User datagram protocol. RFC 768, Information Sciences Institute, University of Southern California, 1980. 3.2.1.2

[19] Jon Postel. Transmission control protocol. RFC 793, Information Sciences Institute, University of Southern California, 1981. 1

[20] Roberto De Prisco, Alan Fekete, Nancy Lynch, and Alex Shvartsman. A dynamic view-oriented group communication service. In *PODC '98: Proceedings of the seventeenth annual ACM symposium on Principles of distributed computing*, pages 227–236, New York, NY, USA, 1998. ACM. 1.1, 1.2, 3

[21] Cristian Tapus, David Noblet, Vlad Grama, and Jason Hickey. MojaveFS: Providing sequential consistency in a distributed objects system. In *ISPDC '06: Proceedings of The Fifth International Symposium on Parallel and Distributed Computing*, pages 66–73, Washington, DC, USA, 2006. IEEE Computer Society. 4

[22] Cristian Tapus, Aleksey Nogin, Jason Hickey, and Jerome White. A mechanism for sequential consistency in a distributed objects system. In *ISCA PDCS*, pages 284–289, 2004. 4.2.1