10-2-2012

Low Latency Fault Tolerance System

Wenbing Zhao
Cleveland State University, w.zhao1@csuohio.edu

P. M. Melliar-Smith
University of California, pmms@ece.ucsb.edu

L. E. Moser
University of California, moser@ece.ucsb.edu

Low Latency Fault Tolerance System

WENBING ZHAO[1], P. M. MELLIAR-SMITH[2,*] AND L. E. MOSER[2]

[1]*Department of Electrical and Computer Engineering, Cleveland State University, Cleveland, OH 44115, USA*
[2]*Department of Electrical and Computer Engineering, University of California, Santa Barbara, CA 93106, USA*
[*]*Corresponding author: pmms@ece.ucsb.edu*

The low latency fault tolerance (LLFT) system provides fault tolerance for distributed applications within a local-area network, using a leader–follower replication strategy. LLFT provides application-transparent replication, with strong replica consistency, for applications that involve multiple interacting processes or threads. Its novel system model enables LLFT to maintain a single consistent infinite computation, despite faults and asynchronous communication. The LLFT messaging protocol provides reliable, totally ordered message delivery by employing a group multicast, where the message ordering is determined by the primary replica in the destination group. The leader-determined membership protocol provides reconfiguration and recovery when a replica becomes faulty and when a replica joins or leaves a group, where the membership of the group is determined by the primary replica. The virtual determinizer framework captures the ordering information at the primary replica and enforces the same ordering of non-deterministic operations at the backup replicas. LLFT does not employ a majority-based, multiple-round consensus algorithm and, thus, it can operate in the common industrial case where there is a primary replica and only one backup replica. The LLFT system achieves low latency message delivery during normal operation and low latency reconfiguration and recovery when a fault occurs.

Keywords: distributed systems; leader–follower replication; membership; message ordering; software fault tolerance; strong replica consistency; virtual synchrony

1. INTRODUCTION

The low latency fault tolerance (LLFT) system provides fault tolerance for distributed applications, using a highly optimized leader–follower replication strategy, to achieve substantially lower latency and more rapid responses than existing group communication systems. LLFT provides fault tolerance for distributed applications over a local-area network, as in a single data center, cluster or cloud, rather than over a wide-area network, such as the Internet.

LLFT provides application-transparent replication, with strong replica consistency, for applications that involve multiple interacting processes or threads. LLFT supports client–server applications where server processes and even client processes are replicated, and multiple-tier applications where middle-tier processes are replicated.

As in other fault tolerance systems, the replicas of a process form a process group. One replica in the group is the primary, and the other replicas are the backups. The primary multicasts messages to a destination group over a virtual connection. The primary in the destination group orders the messages, performs the operations, produces ordering information for non-deterministic operations and supplies that ordering information to its backups. Thus, the backups can perform the same operations in the same order and obtain the same results as the primary. If the primary fails, a new primary is chosen deterministically and the new primary determines the membership of the group.

LLFT operates within the usual asynchronous model, but with timing-based fault detectors. In LLFT, the processing and communication are asynchronous, but the fault detectors impose timing bounds, so that timing faults do indeed occur. The assumptions of eventual reliable communication and sufficient replication enable LLFT to maintain a single consistent infinite computation, despite crash, timing and partitioning faults (but not Byzantine faults).

LLFT uses the leader–follower strategy to establish a total order of messages, to establish a consistent group membership, and to render non-deterministic operations virtually

deterministic. It does not use a majority-based, multiple-round consensus algorithm based on the unreliable failure detectors of Chandra and Toueg [1] to circumvent the impossibility result of Fischer *et al.* [2]. Thus, it can operate in the common industrial case where there are only two replicas (the primary and a single backup), which industry regards as the typical case [3].

1.1. Properties of LLFT

The LLFT system provides fault tolerance for distributed applications within a local-area network, with the following properties.

Strong replica consistency. The LLFT system replicates the processes of an application, and maintains strong replica consistency within an infinite computation. If a fault occurs, LLFT provides reconfiguration and recovery while maintaining virtual synchrony [4, 5], including transfer of state from an existing replica to a new replica and synchronization of the new replica with the existing replicas. The application continues to run without loss of processing or messages, and without disruption to its state. LLFT maintains consistency not only between the replicas in the same group but also between the replicas in different groups that communicate.

Low latency. The LLFT system achieves low latency message delivery during normal operation, and low latency reconfiguration and recovery when a fault occurs. It provides fault tolerance for the applications with minimal overhead in the response times seen by the clients. LLFT achieves low latency by design because, during normal operation, the primary makes the decisions on the order in which operations are performed, and because after a fault the new primary makes the decisions on the new membership and on recovery from the fault. Moreover, the replicated processes of the applications interact with each other directly, without an intermediate daemon process and without additional context switches.

Transparency and ease-of-use. The LLFT system provides fault tolerance that is transparent to the application, for both crash and timing faults but not Byzantine faults. The application is unaware that it is replicated, and is unaware of faults. Applications programmed using transmission control protocol (TCP) socket application programming interfaces (APIs), or middleware such as Java remote method invocation (RMI), can be replicated without modifications to the applications. The application programs require no extra code for fault tolerance, and the application programmers require no special skills in fault tolerance programming. The application program is identical to that of a non-fault-tolerant unreplicated application.

1.2. Architectural components of LLFT

The LLFT system comprises three main architectural components, each of which employs novel techniques. These three components, which are integrated into the complete LLFT system, are described briefly below.

Low latency messaging protocol. The low latency messaging protocol provides reliable, totally ordered, message delivery by communicating message ordering information from the primary replica to the backup replicas in a group. It ensures that, in the event of a fault, a backup has, or can obtain, the messages and the ordering information that it needs to reproduce the actions of the primary. The replicated applications interact with each other directly, via a group multicast.

Leader-determined membership protocol. The leader-determined membership protocol ensures that the members of a group have a consistent view of the membership set and of the primary replica in the group. It effects a membership change and a consistent view more quickly than other membership protocols, by selecting a new primary deterministically, based on the precedences and ranks (defined in Section 2.3) of the backups in the group and by avoiding the need for a majority-based, multiple-round consensus algorithm.

Virtual determinizer framework. The virtual determinizer framework renders the replicas of an application virtually deterministic by recording the order and results of each non-deterministic operation at the primary, and by guaranteeing that the backups obtain the same results in the same order as the primary. The virtual determinizer framework has been instantiated for major sources of non-determinism, including multithreading, time-related operations and socket communication.

1.3. Novel aspects of LLFT

To achieve low latency and strong replica consistency, LLFT makes extensive use of the leader–follower approach. The messaging protocol, the membership protocol and the virtual determinzer framework are all based on the leader–follower approach. Although other researchers [6, 7] have used the leader–follower approach previously, LLFT introduces several novel aspects, which are highlighted below.

Immediate processing and immediate replies. In commercial and industrial distributed systems, one of the most important performance metrics is the latency or response time seen by the client. In LLFT, application messages can be processed by the primary replica in the destination group as soon as it receives them, because the primary determines the message order, unlike consensus-based, total-ordering protocols that require one or more rounds of message exchange before a message can be ordered, delivered and processed. Moreover, in LLFT, application messages can be transmitted immediately, unlike static sequencer or rotating sequencer protocols where a message must wait for sequencing before the message can be transmitted. Immediate processing and immediate replies substantially reduce the latency of the messaging protocol, described in Section 3.

Continued operation during partitioning. An inevitable conflict exists between the requirements of Consistency, Availability and Partitioning, presented by the consistency

availability partitioned theorem [8], which states that a system can provide any two of these three properties but not all three. Some systems exclude the possibility of partitioning. Other systems permit partitioning but ensure consistency by allowing continued operation in only one component of the partitioned system. Unfortunately, such systems sometimes preclude continued operation even though processors are available. In commercial and industrial distributed systems, the risk of partitioning is real and the system must continue to operate even when partitioned. The price that LLFT pays to ensure continued operation and to avoid blocking during partitioning is the risk of inconsistency between processes in different components of the partition. When communication is restored, consistency must be reestablished [9, 10]. Reestablishment of consistency is usually not computationally expensive, but it is application-specific and requires custom application programming.

LLFT continues to operate when the system partitions, with perhaps several components of the partition operating temporarily with no communication between them. Using eventual reliable communication and sufficient replication assumptions (defined in Sections 2.1 and 2.2) and a novel precedence mechanism (defined in Section 2.3), LLFT ensures that, when communication is restored, one of the components dominates the other components, only the primary replica in the dominant component continues to operate as the primary and a single infinite sequence of operations is established for the group, as described in Section 2.6.

Reflection of ordering information. The latency of typical primary-backup systems is adversely affected because the primary replica cannot send the response to the client or the remote group until the response and associated ordering information have been sent to, and acknowledged by, the backup replicas. Sending a response immediately to the client, followed by failure of the primary, might result in the backup replicas being unable to reproduce that response.

LLFT introduces novel mechanisms that allow a primary replica to send application messages directly to remote groups without first sending them to the backup replicas in its local group, which substantially reduces the latency. In LLFT, the primary replica communicates message ordering and other ordering information, needed by the backup replicas in its local group, by piggybacking the ordering information onto a message that it sends to the remote group. At the remote group, the ordering information is reflected back to the backup replicas in the primary's group, by piggybacking the ordering information onto the acknowledgment for the message, as described in Section 3.3. After the failure of the primary replica, when a new primary replica assumes the leadership of the group, it communicates with each remote group to obtain all of the ordering information that contributed to the messages sent to the remote groups by the old primary replica.

Buffer management. LLFT introduces novel group watermark and timestamp watermark mechanisms for buffer management, as described in Section 3.4. A replica must buffer each message that it originates and receives, until it knows that it will no longer need the message, either to retransmit the message in response to a negative acknowledgment or to process the message if the primary replica becomes faulty and it becomes the new primary replica.

Leader-determined fault detection. In LLFT, each backup replica monitors the primary replica to detect faults in the primary. The backups are ordered by rank, and the fault detection timeouts are adjusted so that a backup earlier in rank order detects the failure of the primary before a backup later in rank order, as described in Section 4. This novel timeout mechanism reduces the probability that two backups both claim to be the new primary, if the primary fails.

Leader-determined membership. The LLFT membership protocol, described in Section 4, determines a new primary replica and forms a new membership quickly, unlike multiple-round membership protocols. On detecting that the primary replica is faulty, a backup declares itself to be the new primary and forms a new membership immediately, substantially reducing the delay to resumption of application processing. In addition to faster recovery, the LLFT membership protocol can operate with the primary and only a single backup, which is typical in commercial and industrial systems. In contrast, majority-based, consensus-based membership protocols require multiple backup replicas.

Sanitization of non-deterministic operations. In LLFT, the primary in the destination group produces ordering information for sanitizing non-deterministic operations, and supplies that ordering information to its backups, so that they can perform the same operations in the same order. In particular, the virtual determinizer framework introduces novel data structures: the OrderInfo queue at the primary and, for each operation O, the O. OrderInfo queue at the backups, described in Section 5. These data structures provide a uniform representation for sanitizing different kinds of non-deterministic operations to render the operations virtually deterministic.

2. BASIC CONCEPTS

2.1. System model

LLFT operates in an asynchronous distributed system that comprises one or more applications running on multiple processors and communicating over a local-area network, such as an Ethernet. An application consists of one or more processes, possibly multithreaded with shared data, that interact with each other. Clients that run outside the local-area network are supported via a gateway. Such clients are typically pure clients that are not replicated, whereas the gateway is replicated.

In the underlying asynchronous system, a process that is non-faulty completes a computation, and there is no explicit bound on the time taken to complete the computation. The processes communicate via messages using an unreliable, unordered message delivery protocol, such as user datagram protocol

(UDP) multicast, and there is no explicit bound on the time taken to communicate a message. However, LLFT uses fault detectors that impose implicit timing bounds on the computation and communication in the form of timeouts. The timeouts are local to a processor, and LLFT does not require clocks that are synchronized across multiple processors. Like other researchers [11], we adopt the assumption of *eventual reliable communication*, i.e. if a message is transmitted repeatedly, it is eventually received by the intended destinations.

2.2. Fault model

The LLFT system replicates application processes to protect the application against various types of faults, in particular:

(i) *Crash fault*: A process does not produce any further results.
(ii) *Timing fault*: A process does not produce, or communicate, a result within a timing constraint imposed by the LLFT fault detectors.

LLFT does not handle Byzantine process faults. LLFT allows processes to recover but, when a process recovers, it is regarded as a new process with a new identity (birthId).

LLFT also handles communication network faults, including message loss, communication loss and partitioning faults. Partitioning faults are transient, and the eventual healing of partitioning faults is guaranteed by the eventual reliable communication assumption.

To achieve liveness and termination of the algorithms, LLFT uses fault detectors based on timeouts.[1] The fault detectors are necessarily unreliable, and the timeouts are a measure of how unreliable the fault detectors are. Process crash faults and network partitioning faults are detected as timing faults by the fault detectors. When we say that a process becomes faulty, we mean that it is determined to be faulty by a fault detector.

LLFT does not assume that a majority of the processes in a group are non-faulty, as do [1, 6, 12]. Rather, LLFT adopts the assumption of *sufficient replication*, i.e. in each successive membership of an infinite computation, there exists at least one replica that does not become faulty.

2.3. Process groups

As shown in Fig. 1, the replicas of a process form a *process group (virtual process)*. Each process group has a unique identifier (*group id*). The group id is mapped by LLFT to a virtual port on which the group sends and receives messages, as discussed in Section 2.4.

Each process group has a *group membership* that consists of the replicas of the process. The membership is a subset of a pool of potential members, that changes as faulty processes are removed, repaired and returned to the pool. Typically, different members of a process group run on different processors.

One of the members in a process group is the *primary replica*, and the other members in the group are the *backup replicas*. The primary replica is the member that formed the group.

The *precedence* of a member of a group is determined by the order in which the member joins the group, as described in Section 4. The primary replica has the lowest precedence of any member in the group. The precedences of the backup replicas determine the order of succession to become the new primary, if the current primary becomes faulty.

In addition to the concept of precedence, LLFT also uses the concept of *rank*. The rank of the primary replica is 1, and the ranks of the backup replicas are $2, 3, \ldots$. The ranks determine the fault detection timeouts, as described in Section 4.

Each membership change that introduces a new primary replica constitutes a new *primary view* with a *primary view number*. Each member of a process group must know the primary replica in its group. The members of a sending group do not need to know which member of a destination group is the primary.

2.4. Virtual connections

The LLFT system introduces the novel, elegant idea of a virtual connection, which is a natural extension of the point-to-point connection of TCP.

A *virtual connection* is a communication channel over which messages are communicated between two endpoints, where each endpoint is a process group. A virtual connection is a full-duplex communication channel between the two endpoints. A sender uses UDP multicast to send messages to a destination group over the virtual connection.

A *virtual port (group id)* identifies the source (destination) group from (to) which the messages are sent (delivered) over the virtual connection. All members of a group listen on the same virtual port, and members of different groups listen on different virtual ports. The groups need to know the virtual ports (group ids) of the groups with which they are communicating, just as with TCP.

A process group can be an endpoint of more than one virtual connection, as shown in Fig. 1. Typically, there are multiple process groups, representing multiple applications running on different processors and interacting over the network, but there might be only two process groups and one virtual connection.

2.5. Replication

The LLFT system supports two types of leader–follower replication, namely:

(i) *Semi-active replication*: The primary orders the messages it receives, performs the operations and

[1] According to the authoritative definitions of fault, error and failure [13], these detectors are correctly referred to as fault detectors, despite the common usage of the term failure detectors.

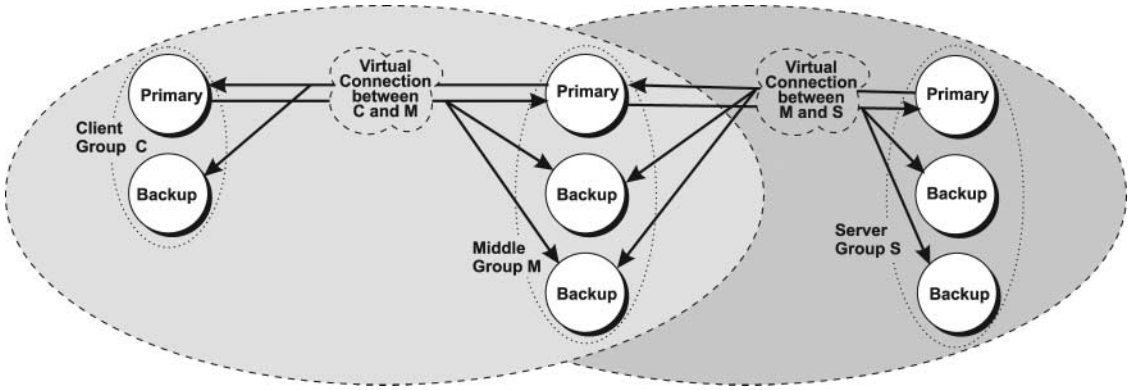

FIGURE 1. Process groups interacting over virtual connections.

provides ordering information for non-deterministic operations to the backups. A backup receives and logs incoming messages, performs the operations according to the ordering information supplied by the primary and logs outgoing messages, but does not send outgoing messages.

(ii) *Semi-passive replication*: The primary orders the messages it receives, performs the operations and provides ordering information for non-deterministic operations to the backups. In addition, the primary communicates state updates to the backups. A backup receives and logs incoming messages, and installs the state updates, but does not perform the operations and does not produce outgoing messages.

Semi-passive replication uses fewer processing resources than does semi-active replication; however, it incurs greater latency for reconfiguration and recovery, if the primary becomes faulty.

To maintain strong replica consistency, it is necessary to sanitize (mask) non-deterministic operations not only for semi-active replication but also for semi-passive replication. For example, consider requests from two clients that are processed concurrently using semi-passive replication. Processing the request from the first client updates a data item. Processing the request from the second client updates the same data item, where the interaction between the processing of the two requests is non-deterministic. The request processing completes, and the primary sends replies to the clients. The primary then fails before it sends its updates to the backups. The processing of the requests from the two clients is repeated at the new primary. However, the non-deterministic interactions between the processing of the two requests is encoded in the replies sent to the clients. The processing of the requests at the new primary must repeat the same non-deterministic interactions, if the correct results are to be obtained.

If a group of replicas becomes partitioned, LLFT ensures that only one component of the partition, referred to as the *primary component* and determined by the precedence of the primary, survives in an infinite sequence of consecutive primary views of the group. Within the primary component, LLFT maintains *virtual synchrony* [4, 5], i.e. if the primary fails, the new primary must advance to the state of the old primary, and the state known to the remote groups of its connections, before the old primary failed. The processes of the other components might terminate operations and must reapply for admission to the membership. Care must be taken to recover those operations and to restore consistency [9, 10]. LLFT pays this price to ensure continued operation and to avoid blocking during partitioning.

2.6. Correctness properties

The novel safety and liveness properties of LLFT, based on the above system model, are stated below. Traditionally, safety and liveness properties are strictly separated. However, in a system that might incur a communication partitioning fault with subsequent recovery from that partitioning, safety necessarily depends on liveness. While the system is partitioned, even complete knowledge of all processes does not suffice to determine which of the competing branches is a transient side branch that will be pruned when the partition is healed. Thus, the safety properties for LLFT are defined in terms of an infinite sequence of consecutive primary views, as assured by the liveness properties. The proofs of correctness can be found in the Appendix. A discussion of the assumptions of the model, relative to these properties, is included below.

2.6.1. Safety properties

For each process group:

(i) At most one infinite sequence of consecutive primary views exists. Each of those consecutive primary views has a unique consecutive primary view number and a single primary replica.

(ii) At most one infinite sequence of operations in an infinite sequence of consecutive primary views exists.

(iii) In semi-active (semi-passive) replication, for a member in a view of the infinite sequence of

consecutive primary views, the sequence of operations (states) of that member is a consecutive subsequence of the infinite sequence of operations (states) of the group.

2.6.2. *Liveness properties*

For each process group:

(i) At least one infinite sequence of consecutive primary views exists.
(ii) At least one infinite sequence of operations in an infinite sequence of consecutive primary views exists.

LLFT imposes implicit bounds on the computation time and the communication time in the form of tunable timeout parameters of the fault detectors. Owing to the asynchrony of the system, those bounds might be violated, which might lead to a replica's being regarded as having incurred a timing fault and being (mistakenly) removed from the membership. The choice of fault detection timeouts is an important design decision. In practice, the fault detection timeout values are determined experimentally under high-load conditions.

With the assumption of eventual reliable communication (i.e. if a message is transmitted repeatedly, it is eventually received), a replica that is mistakenly removed from the membership eventually receives a message indicating that it has been removed. The replica then applies for readmission to the membership, as a new process with a new identity. Without the eventual reliable communication assumption, a mistakenly removed replica might not receive those messages and, thus, might not apply for readmission.

With the assumption of sufficient replication (i.e. each group contains enough replicas such that, in each primary view, there exists a replica that does not become faulty), the sequence of operations of a group is infinite. Without that assumption, the sequence of operations of a group might be finite.

The LLFT system model differs from the models of other researchers with respect to asynchrony. Two examples are Cristian and Fetzer's timed asynchronous distributed systems model [14] and Fetzer's perfect failure detection in timed asynchronous systems model [15]. In LLFT, the processing and the communication are asynchronous, but fault detectors are used to impose timing bounds on the processing and communication, so that timing faults do indeed occur. The eventual reliable communication assumption does not imply a bound on communication delays, but it ensures that transient partitions of the system will be detected. The sufficient replication assumption allows LLFT to define a single consistent infinite computation, despite asynchronous communication delays. In contrast, Cristian and Fetzer [14] define a ΔF stability property to ensure progress, which complicates the pure asynchronous model. Fetzer's model [15] does not require ΔF stability; rather, it employs a majority partition variant or a primary partition variant, neither of which ensures a single consistent infinite computation.

3. LOW LATENCY MESSAGING PROTOCOL

The LLFT messaging protocol converts the unreliable, unordered message delivery service of UDP multicast into a reliable, totally ordered message delivery service between two group endpoints, just as TCP converts the unreliable message delivery service of internet protocol unicast into a reliable, totally ordered message delivery service between two individual endpoints.

The messaging protocol provides the following services for the application messages:

(i) *Reliable message delivery*: All non-faulty members in a group receive each message that is multicast to the group on a connection.
(ii) *Total ordering of messages*: All non-faulty members in a group deliver the messages to the application in the same sequence.
(iii) *Buffer management*: When a message no longer needs to be retransmitted (because the intended destinations have received it), the source and the destinations remove the message from their buffers.

In addition, the messaging protocol helps to maintain virtual synchrony [4, 5] in the event of a membership (view) change when a member joins or leaves the group, either voluntarily or due to the member's failure and subsequent recovery, as described in Section 4. The messaging protocol also incorporates flow control mechanisms to ensure that processing in the primary receiver, and in its backups, can keep up with the primary sender, and that buffer space does not become exhausted. The messaging protocol introduces novel mechanisms, including piggybacking and reflection of ordering information, and group watermarks for buffer management, as described below.

3.1. Data structures

3.1.1. *Message types and message header*

The types of messages used by the messaging protocol are shown at the left of Fig. 2 and are illustrated in Fig. 3. A `Request` or `Reply` message can be either a synchronous blocking request or reply message, or an asynchronous one-way message.

The fields of the message header are shown at the right of Fig. 2. The `msgSeqNum` field is non-zero if and only if the message is a `Request` or `Reply` message. Such messages are inserted into the sent list at the sender and the received list at the destination. The `ack` field acknowledges not only the acknowledged message but also prior messages from the primary.

3.1.2. *Variables*

For each connection, the messaging protocol uses the variables shown at the left of Fig. 4. The message sequence number is used by a member of the destination group to ensure that

Message Types

Request	A message that carries application payload and that is sent by a primary client.
Reply	A message that carries application payload and that is sent by a primary server.
FirstAck	A control message that is sent by a primary to acknowledge the receipt of a `Request` or `Reply` message.
SecondAck	A control message sent by a backup to acknowledge the receipt of a `FirstAck` message.
Nack	A control message sent by a backup to its primary, or by a primary to the primary that originated a missing `Request` or `Reply` message, which then retransmits the message.
Heartbeat	A control message sent by the primary and the backups to facilitate fault detection, and also to share timestamp watermark information which is used for buffer management.
KeepAlive	A control message sent by the primary to a remote group over an inter-group connection, to indicate the liveliness of the connection, after the connection has been idle more than a predetermined amount of time.

Message Header

messageType	The type of message (`Request`, `Reply`, `FirstAck`, `SecondAck`, `Nack`, `Heartbeat`, `KeepAlive`).
sourceGroupId	The identifier of the source group of the message.
destGroupId	The identifier of the destination group of the message.
connSeqNum	A connection sequence number used to identify the connection on which the message is sent.
primaryViewNum	The primary view number, a sequence number that represents the number of membership changes that involve a change in the primary.
precedence	The precedence of the primary.
msgSeqNum	The message sequence number, which is non-zero if and only if the message is a `Request` or `Reply` message multicast by the primary.
ackViewNum	The primary view number for the message whose message sequence number is in the `ack` field.
ack	A message sequence number, which is non-zero if and only if the message is a `Request` or `Reply` message, and the primary has received all messages on the connection with sequence numbers less than or equal to this sequence number.
back	A timestamp watermark used for buffer management to indicate that all members of a group have received all messages with timestamps less than this watermark.
timestamp	A timestamp derived from a Lamport logical clock at the source of the message.

FIGURE 2. The message types and the message header fields used by the Messaging Protocol.

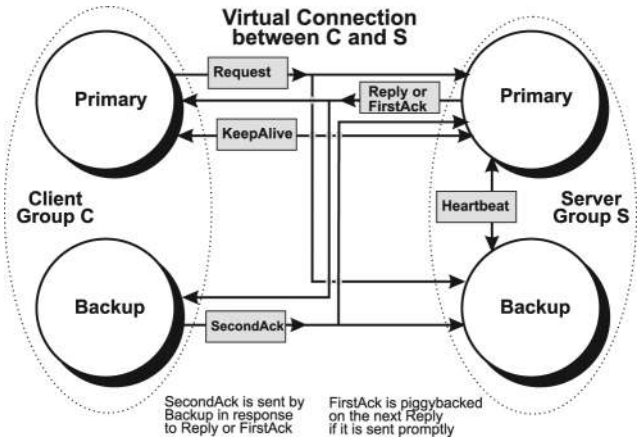

FIGURE 3. Message exchange between a client group *C* and a server group *S*.

it has received all messages from the sending group on the connection. When a member acknowledges the message with sequence number `receivedUpToMsn`, it indicates that it has received all messages with sequence numbers less than or equal to that sequence number. The variables shown at the right of Fig. 4 are discussed in Section 3.4 on buffer management.

Figure 5 shows the variables used for ordering non-deterministic operations, the `OrderInfo` struct and the `MsgOrder` struct. The opaque field stores different `OrderInfo` for different message types. The ordering information is discussed in more detail in Section 5 on the virtual determinizer framework.

3.2. Reliable message delivery

Reliable message delivery requires that all non-faulty members in a group receive each message that is multicast to the group on a connection. The mechanisms that LLFT uses to provide reliable message delivery are the `msgSeqNum`, `ackViewNum` and `ack` fields in the message header and the `FirstAck`, `SecondAck` and `Nack` messages. Reliable message delivery is described below in terms of a `Request` from a client group *C* to a server group *S*, and is illustrated in Fig. 3. The same considerations apply for a `Reply` from a server group *S* to a client group *C*. The pseudocode for the messaging protocol is given in Fig. 6.

The primary in group *C* multicasts messages originated by the application to a group *S* over a virtual connection. It stores the message in the `sent list` for the connection (lines 15–17), and retransmits a message in the `sent list` if it does not receive an acknowledgment for the message sufficiently promptly (as determined by a timeout) (lines 45–46). A backup in group *C* creates and logs (but does not multicast) messages originated by the application. Restricting the actions of the backup in this way reduces the amount of network traffic.

The primary in group *S* includes, in the header (`ack` field) of each application message it multicasts to group *C* on the connection, the message sequence number of the last application message it received without a gap from group *C* on that connection (line 10). If the primary in group *S* does not have a message to multicast sufficiently promptly (as determined by a timeout), it multicasts a `FirstAck` message containing the acknowledgment (lines 47–48).

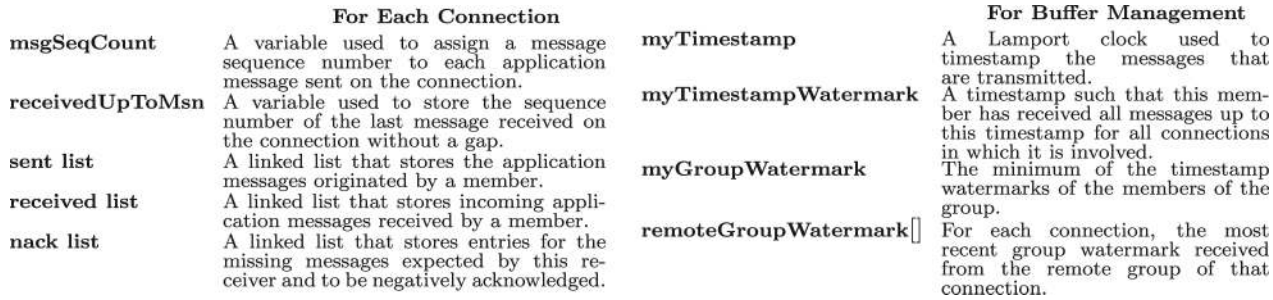

FIGURE 4. Variables used for each connection and global variables used for buffer management.

```
struct OrderInfo
{
  OrderType m_orderType
  SeqNumType m_orderSeqNum
  union
  {
    MsgOrder m_msgO
    MutexOrder m_mtxO
    TimeOrder m_timeO
    SocketOrder m_socO
  }
}
```

```
struct MsgOrder
{
  ViewNumType m_primaryViewNum
  MsgType m_msgType
  ConnSeqNumType m_connSeqNum
  short m_sockFd
  unsigned short m_opaque
  GrpIdType m_remoteGrpId
  SeqNumType m_msgSeqNum
  SeqNumType m_orderSeqNum
}
```

FIGURE 5. Variables used for ordering non-deterministic operations, the `OrderInfo` struct and the `MsgOrder` struct.

On receiving an application message, the primary (or a backup) in group S first checks whether the precedence in the message is greater than the precedence of its own primary. If so, it abandons its current membership, resets its state and rejoins the group membership containing the primary of higher precedence (lines 18–20). Otherwise, the primary (or a backup) in group *S* inserts the application messages it receives on the connection into the `received list` for the connection (lines 25, 31), and updates its `receivedUpToMsn` variable (last message received without a gap) (line 32). If the replica detects a gap in the message sequence numbers (lines 22–25), it creates a placeholder for the missing message, and inserts a corresponding entry into the `nack list`. When the replica receives a retransmitted message and it has a placeholder for the message, it replaces the placeholder with the message and, otherwise, discards the message (lines 26–30).

If a backup in group *C* receives a `FirstAck` message and the backup application has generated the message that the `FirstAck` acknowledges, the backup responds with a `SecondAck` message (lines 60–68). On receiving the `SecondAck` message, the primary in group *S* stops retransmitting the `FirstAck` message (lines 70–71).

If the primary in group *C* receives too many `FirstAck` messages from the primary in group *S*, acknowledging a message that the primary in group *C* sent, then the primary in group *S* has not received a `SecondAck` from the backups in group *C*. Consequently, the primary in group *C* invokes the intra-group flow control mechanisms to slow down, so that the backups in group *C* can catch up (lines 65–66).

The primary (or a backup) in group *S* multicasts a `Nack` message on the connection (lines 49–52), if it determines that it has not received a message from the primary in group *C* on a connection, i.e.

(i) The primary (or the backup) in group *S* sees a gap in the message sequence numbers of the messages it received (line 24) or

(ii) A backup in group *S* receives a `SecondAck` message that contains an `ack` for a message that it has not received (line 72) or

(iii) A backup in group *S* receives a message from the primary in group *S* that orders a message that the backup has not received.

If the primary in group *S* does not have a message to multicast on an inter-group connection sufficiently promptly (as determined by a timeout), it multicasts a `KeepAlive` message to indicate the liveness of the connection (lines 54–55). The primary and each backup in group *S* periodically exchange `Heartbeat` messages on the intra-group connection (lines 56–59), so that each knows that the other has not failed.

3.3. Total ordering of messages

To maintain strong replica consistency among the replicas in a group, all of the replicas must process the same messages in the same order to obtain the same results. In LLFT, that order is determined by the primary replica, thus avoiding the delays associated with multiple-round consensus algorithms. The primary replica must communicate that ordering information, directly or indirectly, to the backup replicas. LLFT ensures that if a primary replica fails and a backup replica becomes the new primary, then the new primary has sufficient ordering information to maintain consistency for each connection between its group and another group.

This requirement that the backup replicas have ordering information is usually ensured by having the primary replica transmit the ordering information to the backup replicas, and by having the backup replicas acknowledge the ordering

```
On sending an application message M
begin
1   msn ← msg seq number assigned to M
2   ack ← msn of the last msg received without a gap
3   wm ← group-wide timestamp watermark
4   ts ← timestamp assigned to M
5   so ← source ordering information
6   ro ← remote ordering information
7   if amPrimary() then
8     record send message order
9   M.setMsgSeqNum(msn)
10  M.setAckField(ack)
11  M.setBackField(wm)
12  M.setTimestamp(ts)
13  M.setPrecedence(precedence)
      // precedence of primary
14  M.piggybackOrderInfo(so,ro)
15  if amPrimary() then
16    multicast M to destination group
17  append M to sent list for retransmission
end

On receiving an application message M
begin
18  if M.precedence > precedence of primary then
    {
19    multicast M on the intra-group connection
20    reset state and rejoin the group
    }
    else
    {
21    msn ← next expected msg seq number
22    if msn < M.getMsgSeqNum() then
      {
23      create placeholders for missing messages
24      append a Nack to nack list
25      append M to received list
      }
26    else if msn > M.getMsgSeqNum() then
      {
27      if M was missing then
        {
28        replace the placeholder with M
29        remove the Nack from nack list
        }
        else
30        discard retransmitted M
      }
      else
31      append M to received list
32    update receivedUpToMsn
33    handle piggybacked ordering information
    }
end

On delivering an application message M
begin
34  M ← first message in received list
35  if not a placeholder for M then
    {
36    if amPrimary() then
37      deliver M and create a received message order
38    if amBackup() then
      {
39      find first msg order
40      if found and msg order orders M then
41        deliver M
      }
42    if M is delivered then
43      move M from received list to delivered list
    }
end

On periodic processing
begin
44  M_sent ← message in the sent list
45  if amPrimary() and M_sent not acked then
46    retransmit M_sent
47  if amPrimary() and SecondAck not received then
48    retransmit a FirstAck for last msg received without a gap
49  if amPrimary() and nack list not empty then
50    retransmit Nack to the remote group
51  if amBackup() and nack list not empty then
52    retransmit Nack to the local group
53  deliver the first ordered message, if any, to the application
end

On expiration of the KeepAlive timer for a connection
begin
54    multicast a KeepAlive message on the connection
55    reset the KeepAlive timer for the connection
end

On expiration of the Heartbeat timer
begin
56  if amPrimary() then
57    multicast a Heartbeat message to the backups
    else // backup
58    transmit a Heartbeat message to the primary
59  reset the Heartbeat timer
end

On receiving a FirstAck message M_FirstAck
begin
60  M ← message in the sent list
61  num_acked ← number of FirstAck received for M
62  MAX_ACK ← max number of FirstAck for M
63  find message M corresponding to M_FirstAck
64  if M is found then
    {
65    if num_acked > MAX_ACK then
66      invoke intra-group flow control
      else
67      if amBackup() or amTheOnlyPrimary() then
68        multicast a SecondAck for M
    }
end

On receiving a SecondAck message M_SecondAck
begin
69  find message M corresponding to M_SecondAck
70  if M is found then
71    M.stopRetransmitFirstAck()
    else
72    append a Nack to nack list
end

On receiving a Nack message M_Nack
begin
73  find message M, received or sent, corresponding to M_Nack
74  if M is found then
75    retransmit M
end

On garbage collecting an application message M
begin
76  ts ← M.getTimestamp()
77  myGrpWatermark ← group watermark
78  remoteGrpWatermark ← remote group watermark
79  if M is on sent list then
80    if ts <= remoteGrpWatermark then
81      remove M from sent list and delete M
82  if M is on delivered list then
83    if ts <= myGrpWatermark then
84      remove M from received list and delete M
end
```

FIGURE 6. Pseudocode for the messaging protocol.

information, before the primary replica transmits a message over a connection to another group. However, doing so causes significant delay and increases the latency of responding to messages between groups.

Instead, in LLFT, the primary replica in group *C* piggybacks on each message it originates and sends on a connection the ordering information for messages it sent and received on the connection since the last message it sent on the connection. It also piggybacks on the message other ordering information, as described in Section 5. When the primary in group *S* receives the ordering information, it reflects the ordering information back to group *C* in its next multicast message. Thus, a backup in group *C* does not receive the ordering information directly from the primary in group *C*, but reflected by the primary in group *S*. The primary in group *C* piggybacks the ordering information on each message it sends until it receives that information reflected back to it.

The piggybacking and reflection mechanisms described above operate in both directions over the connection to ensure that both sets of backup replicas receive ordering information from their own primary replica. These mechanisms allow the primary to send a reply to a client without first sending ordering information or state updates to its own backups, which significantly reduces the latency.

In contrast to existing protocols for consistent message ordering within groups [4, 16–18], LLFT does not maintain an order that is stable across the replicas in a group. Instead, LLFT maintains consistency by having the primary replica determine the order, while it continues to operate. If the primary replica fails, the new primary replica determines the order, based on ordering information obtained from the primary replicas of the other groups. That ordering information extends the history of the group in a manner that is consistent with the information held by the other groups.

3.4. Buffer management

A replica in a group must retain each message that it originates and receives, until it knows that it will no longer need the message, either to retransmit the message in response to a negative acknowledgment or to process the message if the primary becomes faulty and it becomes the new primary.

LLFT uses novel group watermark and timestamp watermark mechanisms for buffer management. The `myTimestampWatermark` variable contains the minimum of the timestamps of the messages that the primary or a backup received on all of its connections. A backup puts the value of the `myTimestampWatermark` variable into the `back` field of a control message that it sends to its primary. The `myGroupWatermark` variable contains the minimum timestamp watermark of the group, i.e. the minimum of the primary's own `myTimestamp Watermark` and all of its backups' `myTimestampWatermarks`. The primary puts the value of `myGroupWatermark` into the `back` field of a message that it multicasts. The primary (a backup) in a group maintains an array `remoteGroupWatermark[]` that stores the latest group watermarks received from the remote groups of its connections.

As shown in Fig. 6 (lines 76–84), a replica that sends a message on a connection garbage-collects the message if the `timestamp` in the message header is less than or equal to the `remoteGroupWatermark`. A replica that receives and delivers a message garbage-collects the message if the `timestamp` in the message header is less than or equal to the `myGroupWatermark` for that replica's group.

4. LEADER-DETERMINED MEMBERSHIP PROTOCOL

LLFT addresses the problem of determining a new membership set for a process group, when a fault occurs or when a member joins or leaves the group, and the problem of maintaining a consistent view of the membership. Other existing membership protocols [6, 12] use a two-phase commit algorithm with a majority of replicas that vote for a membership, to achieve consensus agreement and to avoid a split-brain situation in which competing memberships are formed.

In the presence of unreliable communication that results in network partitioning, it is difficult or expensive to eliminate the risk of competing memberships. If partitioning occurs, some of the members might form a new membership, while other members continue to operate with the existing membership. This situation can be avoided only if *every* value communicated is subjected to a majority vote of the members, as is done in aircraft flight control systems such as SIFT [19]. Under conditions of unreliable communication and partitioning, it is undesirable to degenerate into multiple competing memberships, but it is also undesirable to fail to form a membership. Our objectives are to ensure that a membership is formed, to detect partitions and to reestablish a consistent state when the partition heals.

The LLFT leader-determined membership protocol makes a deterministic choice of the new primary replica. The new primary replica determines the addition (removal) of the backups to (from) the membership, and their precedences and ranks, as described below. Thus, it operates faster than a multiple-round consensus algorithm [1], which is important because application processing is suspended while the new membership is being formed. The membership protocol ensures that all of the members have the same primary, the same membership set and the same primary view number.

The *precedence* of a member of a group is the order in which the member joins the group. If a member becomes faulty and later rejoins the group, it joins as a new member and, thus, has a new precedence. The primary adds a new member to the membership as a backup, and assigns to the backup the next precedence in the sequence for its primary view. Thus, a backup added later to the primary view has a higher precedence. To ensure unique precedence values, even if two backups are admitted to two distinct primary views while the system is transiently partitioned, the precedence of

a new backup is qualified by the precedence of the primary. Consequently, the precedence of a member of a group is actually a sequence of precedence numbers. Theoretically, such a sequence could become lengthy but, in practice, it does not, because the sequence can be pruned when the primary determines that potential members of the group are indeed members. The precedence is important because it determines the order of succession of the backups to become the new primary, if the current primary becomes faulty.

The *rank* of the primary replica is 1, and the ranks of the backup replicas are $2, 3, \ldots$. When a new primary forms a new membership or adds a new backup to the membership, it assigns ranks to the backup(s) in the order of their precedences. The rank of a member can change when another member is removed from the group, whereas the precedence of a member is assigned when it joins the group and does not change while it is a member. The ranks of the members are consecutive, whereas the precedences might not be.

The ranks determine the timeouts for detection of faults in the primary and the backups. The backup with rank 2 operates a fault detector to determine that the primary is faulty. The backup with rank 3 operates a fault detector to determine that the primary is faulty and also that the backup with rank 2 is faulty, because the backup with rank 2 did not determine that the primary is faulty.

To reduce the probability of a race condition in which two backups both claim to be the next new primary, the fault detection timeouts for the backups increase with increasing rank, a novel technique introduced by LLFT. For example, the timeout of the fault detector of the backup with rank 2 might be 10 ms to allow time for the backup to detect that the primary has become faulty, whereas the timeout of the fault detector of the backup with rank 3 might be longer, say 30 ms, allowing 10 ms of inaction by the primary, 10 ms of inaction by the backup with rank 2 and an additional 10 ms for skew between the timeouts of the two backups. The successively longer fault detection timeouts for backups of higher ranks make it quite unlikely that their timeouts will expire, unless all of the lower rank backups are indeed faulty. However, it might still happen that two backups both propose to become the new primary.

The fault detection timeouts must be chosen carefully. Timeouts that are too long cause unnecessary delays after a fault, whereas timeouts that are too short cause membership churn and readmission of members to the group, which can also increase latency. In practice, appropriate timeout values are determined experimentally under high-load conditions.

If the system is not partitioned, the backup with lower precedence detects that there is a membership whose primary has higher precedence. The backup with lower precedence gives up and the backup with higher precedence continues. For example, if the backup with rank 2 and the backup with rank 3 both propose to become the new primary, the backup with rank 3 overrides the backup with rank 2 because the backup with rank 3 has higher precedence.

If the system is partitioned, LLFT continues to operate transiently in two (or more) components of the partition, instead of blocking operations in some components as would occur in a strict primary component model. When communication is restored, one component will dominate the other component(s) because the precedence of its primary is higher. The operations, performed while the system is partitioned, must be reconciled. The dominated component is then pruned, and the dominant component continues to operate as the primary, with reconciliation of states as described below.

Reconciliation of states following the partition is addressed in prior work [9], which merges the sequences of write operations from the various components of the partition. Similarly, our work [10] on fulfillment transactions within the dominant component repeats the operations of the dominated component, with automatic reconcilation of inconsistencies in simple cases and a framework for handling more difficult cases in which the components performed inconsistent actions. In contrast, while the system is partitioned, the Bayou system [20] provides tentative and committed write operations and application-specific reconciliation of conflicting writes. In all of these approaches, compensation operations might affect other groups, directly or transitively.

Membership changes that correspond to a change of the primary constitute a view change, referred to as a *primary view change*. When the primary view changes, the proposed new primary adjusts the members' ranks, and resets the message sequence number to one on each of its connections.

The backups must change the primary view at the same virtual synchrony point as the primary. To this end, the new primary produces ordering information for the primary view change and multicasts that ordering information to the backups. A backup changes to the new primary view when it has received all of the messages, performed all of the operations that were ordered before the virtual synchrony point and reached the same state, as described in Section 4.2.2. Establishing a virtual synchrony point also serves to reduce the risk that two (or more) backups regard themselves as the new primary.

4.1. Data structures

4.1.1. Message types

The types of messages used by the membership protocol are described in Fig. 7, and are illustrated in Fig. 8.

The `ProposePrimary`, `ProposeBackup`, `AcceptBackup` and `RemoveBackup` messages are multicast on the intra-group connection.

The `ProposePrimary`, `AcceptBackup` and `RemoveBackup` messages include the old membership in the payload, and require an explicit acknowledgment from each backup. For the primary, these acknowledgment messages serve as 'commit' messages. The new or existing primary must retransmit these messages until all of the backups in the membership (as determined by the primary) have acknowledged them.

Message Types for Primary Change	
ProposePrimary	A message multicast by a self-appointed new primary to request a change of the primary.
NewPrimaryView	A message multicast by the new primary on each of its connections (other than the intra-group connection) to report the new primary view and to collect information regarding the old primary.

Message Types for Backup Change	
ProposeBackup	A message multicast by a new replica that wants to join the group.
AcceptBackup	A message multicast by the primary to add a backup to the group.
RemoveBackup	A message multicast by the primary to remove a backup from the group.
State	A message sent by the primary to a backup, containing the checkpointed state of the primary.

FIGURE 7. The types of messages used by the Membership Protocol for change of the primary or a backup.

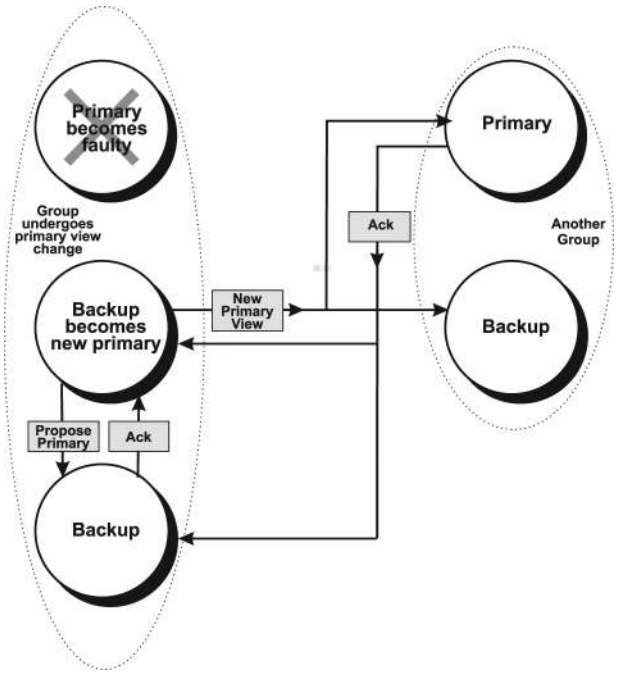

FIGURE 8. Message exchange when a primary view change occurs.

4.2. Change of the primary

The change of the primary in a group is handled in two phases, as discussed below. The pseudocode for the membership protocol for a change of the primary is shown in Fig. 9. In the rules below, V_i denotes the primary view with primary view number i which corresponds to `myPvn` in the pseudocode, and p denotes the precedence of the primary which corresponds to `myPrecedence` in the pseudocode.

4.2.1. Determining the new membership

In the first (*election*) phase, the new primary is determined. The new primary then determines which backups are included in the new membership, as well as their precedences and ranks. More specifically, the first phase operates as follows:

(i) If a backup with precedence p does not receive a `Heartbeat` message from the primary of view V_i within a given time period (and, thus, determines that the primary is faulty) and it has not received a `ProposePrimary` message for view V_i from a backup with precedence $< p$, the backup multicasts a `ProposePrimary` message on the intra-group connection, denouncing the old primary and appointing itself as the new primary of view V_{i+1}.

(a) The backup excludes from the new membership the old primary and the backups of the old membership with precedences $< p$ (line 4). It excludes such a backup because that backup did not send a `ProposePrimary` message quickly enough to become the new primary and, thus, it declares that backup to be faulty.
(b) The backup includes, in the `ProposePrimary` message, the group identifier, the proposed new membership, its current primary view number i and its precedence p (line 5).

(ii) If a backup with precedence q receives a `Propose-Primary` message for a new primary view V_{i+1}, from a proposed new primary with precedence p and the backup is included in the proposed new membership (which implies that $q > p$), and

(a) the backup has not generated a `Propose-Primary` message for view V_{i+1} and
(b) the backup has not acknowledged a `Propose-Primary` message from a backup with precedence $> p$ for view V_{i+1},

then the backup with precedence q accepts the proposed new membership and acknowledges the `ProposePrimary` message (lines 21–24).

(iii) If a backup receives a `ProposePrimary` message for a new primary view V_{i+1}, or a subsequent view, with precedence p and the backup is not included in the proposed new membership, and

(a) the backup has not generated a `Propose-Primary` message for view V_{i+1} and $q > p$ and
(b) the backup with precedence q has not received a `Propose Primary` message for view V_{i+1} from a backup with precedence $> p$,

```
On expiration of the fault detection timer for
the primary(at a backup)
begin
1   myPvn ← current primary view number
2   myPrecedence ← precedence assigned to this member
3   if not received a ProposePrimary message M such that
        M.pvn >= myPvn and M.precedence < myPrecedence then
    {
4     exclude members with lower precedences than myPrecedence
5     multicast a ProposePrimary message Mp with the group id,
          the new membership, myPvn and myPrecedence
6     start retransmission timer
7     retransmissionCount ← 0
    }
  end

  On expiration of the retransmission timer (at the backup
  that sent the ProposePrimary message Mp)
  begin
8   if retransmissionCount > MAX_COUNT then
    {
9     exclude members that have not yet acknowledged my
          ProposePrimary message Mp
10    transmit a ProposePrimary Mp with the latest membership
11    retransmissionCount ← 0
    }
12  restart retransmission timer
13  retransmissionCount++
  end

  On receiving an ack for the ProposePrimary message Mp
  (at the backup that sent Mp)
  begin
14  if received acks from all backups in membership then
    {
15    cancel retransmission timer for ProposePrimary message Mp
16    start recovery protocol
    }
  end

  On receiving a ProposePrimary message Mp (at a backup
  that did not send Mp)
  // primaryPrecedence initially set to precedence of
  // primary in current view
  begin
17  myPvn ← current primary view number
18  if Mp.pvn >= myPvn then
19    if I am in the membership then
      {
20      if Mp.precedence > primaryPrecedence then
        {
21        primaryPrecedence ← Mp.precedence
22        send acknowledgment for ProposePrimary message
23        update membership and ranks
24        start fault detection timer for the new primary
        }
      }
      else
25      reset state and rejoin the group
  end

  On recovering from a primary change (at the new primary)
  begin
26  myPvn++
27  primaryPrecedence ← myPrecedence
28  for each connection do
29    multicast a NewPrimaryView message
  end

  On receiving an ack for the NewPrimaryView message
  Mv (at the new primary)
  begin
30  nack all missing messages until received them
31  retrieve order info held at remote groups
        from application message or KeepAlive message
32  if received all missing messages and reproduced
        all messages sent by the old primary then
    {
33    reset msgSeqNum to 0 on each connection
34    adjust ranks of backups
35    record an order for the primary view change
    }
  end

  On receiving a NewPrimaryView message Mv (at the
  primary of a remote group)
  begin
36  recvUpToMsn ← seq num of last msg received without a gap
37  lastSentMsn ← seq num of last msg sent
38  discard all messages received after recvUpToMsn
39  expectedPvn ← Mv.pvn
40  expectedMsn ← 0
41  acknowledge Mv with (recvUpToMsn, lastSentMsn)
  end
```

FIGURE 9. Pseudocode for the membership protocol to handle the change of the primary.

then the backup resets its state and rejoins the group (line 25).

(iv) When the new primary has received acknowledgments for its `ProposePrimary` message from all members in the proposed new membership, it concludes the first (election) phase and proceeds to the second (recovery) phase (lines 14–16). The new primary sets its new primary view number on line 26.

Note that the sets of conditions in the second and third bullets above are not complementary and collectively exhaustive. If a backup receives a `ProposePrimary` message that does not satisfy either of these sets of conditions, it ignores that `ProposePrimary` message. These novel mechanisms determine the new membership of the group using only one round of message exchange (`ProposePrimary` and corresponding acknowledgments). With the aims of simplicity and timeliness, the mechanisms do not attempt to form a new membership with the largest possible number of members, as do other systems [17, 21].

4.2.2. *Recovering from the membership change*

In the second (*recovery*) phase, the new primary queries the remote group of each of its inter-group connections regarding the old primary's state, and determines a virtual synchrony point. The virtual synchrony point must be consistent with the messages that were sent to remote groups in the prior view, and with the ordering that was used to generate such messages. Thus, the new primary needs to know the last message sent by the old primary and delivered to each remote group on a connection and, in particular, the ordering information piggybacked onto that message. To advance to the state of the old primary known to the remote groups before the old primary became faulty, the new primary must follow the ordering information. More specifically:

(i) The new primary collects information for the virtual synchrony point by multicasting a `NewPrimaryView` message on each of its inter-group connections (lines 28–29). The `NewPrimaryView`

message contains the most recent ordering information known to the new primary for the connection.

(ii) On receiving the `NewPrimaryView` message, the primary of the remote group flushes all messages after the last message delivered from the old primary's group (line 38). The primary of the remote group acknowledges the `NewPrimaryView` message by providing information regarding the last message delivered from, and sent to, the old primary's group (line 41). The primary of the remote group reflects back the ordering information to the new primary either in a new application message, or in a `KeepAlive` message if it does not have an application message to send.

(iii) On receiving an acknowledgment from the primary of the remote group, the new primary determines whether it has missed any messages from that primary. The new primary then sends `Nack` messages for the missing messages until it has received them (line 30). The new primary retrieves the ordering information piggybacked on application messages or `KeepAlive` messages from the primary of the remote group.

(iv) When the new primary has executed all of the operations according to the ordering information determined by the old primary, it concludes the second phase by resetting the message sequence numbers to one, adjusting the backups' ranks and generating ordering information declaring the start of a new primary view (lines 33–35). The backups switch to the new primary view when they receive and process that ordering information.

4.3. Addition or removal of a backup

The membership protocol also addresses the addition or removal of a backup, as shown in the pseudocode in Fig. 10. The pseudocode for the addition of a backup (lines 1–14) includes the case where a process is the first member of the group and, thus, is the primary.

4.3.1. Addition of a backup

A new process begins to log messages when it starts up (line 1). The `myBirthId` of a process (line 5) is a unique identifier, like a birth certificate, that identifies a process wishing to join the membership that does not yet have a precedence. The process multicasts a `ProposeBackup` message on the intra-group connection (line 7). The primary assigns the precedence and rank of the new backup (line 23) and then multicasts an `AcceptBackup` message (line 30), containing the new membership, on the intra-group connection. A backup that receives an `AcceptBackup` message, with a membership containing itself, accepts the new membership and responds with an acknowledgment (lines 15–18).

The primary checkpoints its state when it has received acknowledgments for the new membership from all of the backups in the group (lines 39–41). The checkpoint provides the virtual synchrony point for adding the new backup. The primary transmits the checkpoint to the new backup in a `State` message (line 42). The new backup then sets its state by applying the checkpoint, and replaying the messages from the log (lines 20–21), after deleting obsolete messages.

4.3.2. Removal of a backup

The primary modifies the ranks of the backups in the group (line 28) and then multicasts a `RemoveBackup` message (line 30), containing the new membership, on the intra-group connection. When a backup receives a `RemoveBackup` message that includes itself in the membership, the backup accepts the new membership and responds to the primary with an acknowledgment (lines 43-45). When a backup receives a `RemoveBackup` message that does not include itself in the membership, the backup resets its state and multicasts a `ProposeBackup` message requesting readmission to the membership (line 46).

For both addition and removal of a backup, the primary multicasts the new membership to all of the backups in the membership (line 30). It commits the membership change when it has collected acknowledgments from all of the backups in the membership (line 39). If a backup does not provide an acknowledgment promptly, the primary removes the backup from the membership (line 34).

5. VIRTUAL DETERMINIZER FRAMEWORK

A reliable, totally ordered, message delivery protocol ensures consistent replication only if the application is deterministic. However, modern applications are typically non-deterministic in a number of ways. To maintain strong replica consistency, it is necessary to sanitize or mask such sources of non-determinism, i.e. to render the application *virtually deterministic*.

The LLFT virtual determinizer framework provides a generic algorithm and uniform data structures for sanitizing the sources of non-determinism in an application in a transparent manner. We describe the data structures and algorithms below, as well as communication of ordering information to the backup replicas. First, we describe the threading model.

5.1. Threading model

The state of an application process is determined by data shared among different threads, and by thread-specific data local to a thread.

Each thread within a process has a unique thread identifier. Following good programming practice, every operation on a data item that is shared by multiple threads must be protected by a mutex, even atomic operations such as reads and increments,

```
On joining a process group (at a new backup)
begin
1   start logging
2   hostId ← host id of the joining process
3   pid ← process id of the joining process
4   ts ← local start up time of the joining process
5   myBirthId ← (hostId, pid, ts)
6   Mp ← ProposeBackup message with myBirthId
7   multicast ProposeBackup message Mp
8   start retransmission timer
9   retransmissionCount ← 0
end

On expiration of the retransmission timer for the
ProposeBackup message Mp (at the new backup)
begin
10  if retransmissionCount > MAX_COUNT then
11    become the first member of the group and thus the primary
    else
    {
12    retransmit ProposeBackup message Mp
13    retransmissionCount++
14    restart retransmission timer
    }
end

On receiving an AcceptBackup message Ma (at new backup)
begin
15  if Ma.birthId == myBirthId then
    {
16    accept membership, precedence, rank
17    cancel retransmission timer for ProposeBackup message Mp
18    acknowledge AcceptBackup message Ma indicating
          the need for a state transfer
19    wait for a State message
    }
end

On receiving a State message (at the new backup)
begin
20  restore state
21  replay messages from the log
end

On receiving a ProposeBackup message Mp (at the primary)
begin
22  if ProposeBackup message Mp is not a duplicate then
    {
23    assign precedence and rank
24    add the backup to the membership
25    execute commit protocol code
26    start fault detection timer for the new backup
    }

end

On expiration of the fault detection timer for
a backup (at the primary)
begin
27  remove the backup from the membership
28  adjust ranks of the other backups
29  execute commit protocol code
end

On committing a new membership (at the primary)
begin
30  multicast a membership change message
      // AcceptBackup for adding a backup
      // RemoveBackup for removing a backup
31  start retransmission timer
32  retransmissionCount ← 0
end

On expiration of the retransmission timer for the
membership change message (at the primary)
begin
33  if retransmissionCount > MAX_COUNT then
    {
34    exclude members that have not yet acknowledged
          membership change message
35    retransmit membership change message
          with latest membership
36    retransmissionCount ← 0
    }
37  restart retransmission timer
38  retransmissionCount++
end

On receiving an ack for the membership change
message (at the primary)
begin
39  if received acks for the membership change message from
        all backups in membership then
    {
40    cancel retransmission timer for membership change message
41    get checkpoint of the state
42    send State message to the backup
    }
end

On receiving an AcceptBackup / RemoveBackup
message (at an existing backup)
begin
43  if I am in the membership then
    {
44    update the membership and ranks
45    send acknowledgment to primary
    }
    else
46    reset state and rejoin the group
end
```

FIGURE 10. Pseudocode for the membership protocol to handle the addition and removal of a backup.

because the mutex serves also to reproduce a deterministic order for such operations. The threads and mutexes can be created and deleted dynamically.

Each replica in a process group runs the same set of threads. A thread interacts with other threads, processes and its runtime environment through system/library calls. Non-determinism can arise from different orderings of, and different results from, such calls at different replicas in the group.

If the operations on the shared and local data in different replicas are controlled in such a way that (1) the updates on a data item occur in the same order with the same change and (2) each thread updates the data items in the same order with the same change, then the replicas will remain consistent.

Figure 11 at the left gives an example of the pseudocode for a thread that shows how such calls might change the state of an application. The example illustrates three kinds of system/library calls:

(i) Calls that try to acquire a mutex (line 18). The `pthread_mutex_trylock()` operation is similar to a non-blocking read in that, if the mutex is currently held by another thread, the call returns immediately with a specific error code, so that the caller thread is not blocked. If the thread of one replica successfully claims the mutex, while the corresponding thread of another replica fails, the two replicas perform different operations (lines 19–22), causing divergence of their states, because one replica changes the shared data `SD1` (line 20) while the other changes the thread-local data `LD5` (line 22).

```
1   get current time
2   // update thread-local data LD1
3   do a non-blocking read from socket fd
4   if picked up a message then
    {
5     // update thread-local data LD2
6     handle the message
7     do a non-blocking write to socket fd
8     if failed to write the response then
      {
9       // update thread-local data LD3
10      append to a queued message, if any
      }
    }
    else
    {
11    // update thread-local data LD3
12    flush queued message, if any, to socket fd
    }
13  get current time
14  // update thread-local data LD1
15  if timed out then
    {
16    // update thread-local data LD4
17    call timeout handling routine
    }
18  try to claim mutex Mtx
19  if claimed mutex Mtx then
    {
20    change shared data SD1
21    release mutex Mtx
    }
    else
22    update thread-local data LD5
```

```
On returning from a call (at the primary)
begin
23  T ← thread identifier
24  O ← operation identifier
25  N ← operation count
26  D ← operation metadata
27  OrderInfo ← global queue to store order info
28  append an entry (T, O, N, D) to OrderInfo
end

On receiving an order info entry (T, O, N, D)
(at a backup)
begin
29  if O.OrderInfo does not exist then
30    create O.OrderInfo
31  append (T, N, D) to O.OrderInfo
32  T1 ← first entry in O.OrderedInfo
33  wake up T1 if it is blocked
end

On intercepting a call (at a backup)
begin
34  T1 ← identifier of the thread performing the call
35  O1 ← operation identifier of the call
36  N1 ← count for O1 for any thread
37  get first entry (T, N, D) of O1.OrderInfo
38  while (T, N, D) not available or T1 != T or N1 != N do
39    suspend T1
40  consume (T, N, D) and remove it from O1.OrderInfo
41  return
end
```

FIGURE 11. At the left, the pseudocode for a thread. The system/library calls that might change the state, or lead to a state change, are highlighted in bold. At the right, the pseudocode for the virtual determinizer framework to render the application virtually deterministic.

(ii) Calls that retrieve local clock values (lines 1, 13). These calls change thread-local data (`LD1`) directly (lines 2, 14). If different replicas obtain different clock values, the replicas might make different decisions (line 15) as to whether a timeout occurred. If one replica times out while the other does not, the states of the replicas will diverge because of the difference in thread-local data `LD4` (line 16).

(iii) Calls that read (write) from (to) a socket asynchronously (lines 3, 7, 12). If, for the same read (write) operation, one replica successfully reads (writes) a message while the other does not, the states of the two replicas will differ in the thread-local data `LD2` (line 5) and potentially `LD3` (lines 9, 11).

5.2. Generic algorithm and data structures

Figure 11 at the right shows the pseudocode for the virtual determinizer framework. The virtual determinizer framework records the ordering information and the return value information of non-deterministic system/library calls at the primary to ensure that the backups obtain the same results in the same order as the primary. In particular, for each non-deterministic operation, the virtual determinizer framework records the following information:

(i) *Thread identifier*: The identifier of the thread that is carrying out the operation.

(ii) *Operation identifier*: An identifier that represents one or more data items that might change either during the operation or on completion of the operation.

(iii) *Operation count*: The number of operations carried out by a thread for the given operation identifier.

(iv) *Operation metadata*: The data returned from the system/library call. These metadata include the `out` parameters (if any), the return value of the call and the error code (if necessary).

The virtual determinizer framework introduces novel generic data structures: the `OrderInfo` queue at the primary and, for each operation `O`, the `O.OrderInfo` queue at the backups. These data structures provide a uniform representation for handling different kinds of non-deterministic operations.

At the primary, the `OrderInfo` queue contains four-tuples `(T, O, N, D)`, where thread `T` has executed a call with operation identifier `O` and with metadata recorded in `D`, and this call is the `N`th time in its execution sequence that thread `T` has executed such a non-deterministic call. The `OrderInfo` queue spans different threads and different operations. The algorithm appends a `(T, O, N, D)` entry to the `OrderInfo` queue on return of the operation `O` (lines 23–28). The entries are transmitted to the backups, using the novel piggybacking and reflection mechanisms described in Section 3 for the messaging protocol.

At a backup, for each operation O, the O.OrderInfo queue contains three-tuples (T, N, D), in the order in which the primary created them. When the backup receives the first entry (T, O, N, D) for operation O, it creates the O.OrderInfo queue (lines 29–30). The algorithm then awakens the first thread in the O.OrderInfo queue if it is blocked (lines 31–33). When thread T tries to execute operation O as its Nth execution in the sequence, if (T, N, D) is not the first entry in the O.OrderInfo queue, the algorithm suspends the calling thread T (lines 34–39). It resumes a suspended thread T in the order in which (T, N, D) occurs in the O.OrderInfo queue, rather than the order in which the thread was suspended or an order determined by the operating system scheduler. It removes an entry (T, N, D) from the O.OrderInfo queue immediately before it returns control to the calling thread T after its Nth execution in the sequence (lines 40–41). The algorithm requires the ordering of all related operations, e.g. both claims and releases of mutexes.

Thus, the virtual determinizer framework provides a generic algorithm and data structures for sanitizing different kinds of non-deterministic operations. We describe below how it is instantiated for multithreading, time-related operations and socket communication. We have not yet instantiated the virtual determinizer framework for operating system signals and interrupts, which constitutes future work.

5.3. Multithreading

The Consistent Multithreading Service (CMTS) allows concurrency of threads that do not simultaneously acquire the same mutex. Thus, the CMTS achieves the maximum possible degree of concurrency, while maintaining strong replica consistency.

At the primary, the CMTS creates mutex ordering information, where the `operation identifier` is the mutex Mtx. For the normal mutex claim call (`pthread_mutex_lock()` library call), the `operation metadata` is empty if the call is successful and, otherwise, is the return value. For the non-blocking mutex claim call (`pthread_mutex_trylock()` library call, the `operation metadata` is the return value.

At a backup, to process a mutex ordering information entry, the CMTS examines the metadata. If the metadata contain an error code, the CMTS returns control to the calling thread with an identical error status without performing the call. Otherwise, the CMTS delegates the mutex claim operation to the original library call provided by the operating system. If the mutex is not currently held by another thread, the calling thread acquires the mutex immediately. Otherwise, the calling thread is suspended and subsequently resumed by the operating system when the thread that owns the mutex releases it.

5.4. Time-related operations

The Consistent Time Service (CTS) ensures that clock readings at different replicas are consistent for time-related system calls, such as `gettimeofday()` and `time()`. At the primary, the CTS creates time ordering information, where the `operation identifier` is the time source and the `operation metadata` is the clock value, or an error code if the call fails.

With the CTS, the replicas see a *virtual group clock* that resembles the real-time clock. Each replica maintains an offset to record the difference between its local physical clock and the virtual group clock. Each backup updates its offset for each clock reading.

In addition to consistent clock readings, the CTS ensures the monotonicity of the clock as seen by the replicas in a group, even if the primary is faulty. The new primary must not include its local physical clock value in the time ordering information it sends to the backups, because doing so might roll backward, or roll forward, the virtual group clock. Instead, the new primary adds the recorded offset to its local physical clock value and includes that value in the time ordering information it sends to the backups.

5.5. Socket communication

The Consistent Socket Communication Service (CSCS) produces ordering information for non-blocking read (write) system calls that an application uses to receive (send) messages on a socket asynchronously. If no message is received (sent), the non-blocking read (write) call returns a specific error code. On such an error return, the application might switch to some other task and change to a different state. Thus, the CSCS orders the event of failing to receive (send) a message.

At the primary, on return from a read (write) system call on a socket, the CSCS produces a socket ordering information entry for that operation. The `operation identifier` is the socket file descriptor. The `operation metadata` is an identifier for the message being read (written) if the read (write) succeeds, or an error code if it fails.

It is quite common to combine socket read (write) system calls with select/poll system calls. Typically, the application performs a read (write) system call only if the select/poll system call indicates that the corresponding socket is readable (writable). The select/poll system call offers a timeout parameter (in Linux) for the user to specify how long the operating system can take to return from the call.

At the primary, on return from a select/poll system call, the CSCS produces a socket ordering information entry. The `operation identifier` is the socket file descriptor. The `operation metadata` contains the number of events, the read (write) error mask and the amount of time left before the timeout (in Linux) if the call returns successfully, or an error code if it fails.

6. IMPLEMENTATION AND PERFORMANCE

The LLFT system has been implemented in the C++ programming language for the Linux operating system. The library interpositioning technique is used to capture and control the application's interactions with its runtime environment. The application state is checkpointed and restored using a memory-mapped checkpoint library, derived from [22], that checkpoints the entire address space of an application process. The implementation of LLFT is compiled into a shared library. The library is inserted into the application address space at startup time using the `LD_PRELOAD` facility provided by the operating system. LLFT is transparent to the application being replicated and does not require recompilation or relinking of the application program.

The experimental testbed consists of 14 HP blade servers, each equipped with two 2GHz Intel Xeon processors running the Linux Ubuntu 9.04 operating system on a 1 Gbps Ethernet. A two-tier client group/server group application is used to benchmark the LLFT implementation. The application state is small and the time to communicate it is not significant.

The performance evaluation of LLFT focuses on three areas: (1) the performance of the messaging protocol during normal fault-free operation, (2) the performance of the membership protocol during fault recovery and (3) the overhead of the virtual determinizer framework. As a baseline for comparison of the end-to-end latency and of the throughput of LLFT, we use TCP with no replication. We also compare the end-to-end latency for LLFT with three-way replication and that for Totem [17] with three-way replication. Note that LLFT is designed for low latency, whereas Totem is designed for high throughput.

In all latency measurements during normal operation, 10 000 samples are taken in each run. For the throughput measurements, a sample is taken at the primary replica for every 1000 requests processed (i.e. the sample reflects the mean throughput over 1000 requests), and 10 samples are taken for each run. For the recovery latency measurements, 100 samples are taken and the mean values are reported.

6.1. Messaging protocol

First, we consider the performance of the messaging protocol during normal fault-free operation. We characterize the end-to-end latency in the presence of a single client for various message sizes: (1) short requests and short replies, (2) various size requests and short replies and (3) short requests and various size replies. The end-to-end latency for (2) is virtually indistinguishable from that for (3) for the same message sizes (for requests and replies). Consequently, we consider only the message size here.

Figure 12 shows the mean end-to-end latency as a function of message size, without replication using TCP, and with three-way replication using LLFT. Error bars, corresponding to the standard deviation, are shown for LLFT. As the figure shows, the messaging protocol incurs very moderate overhead, ranging from ∼15% for large messages to ∼55% for small messages, caused primarily by the piggybacking of ordering information. For large messages, which require fragmentation in user space, the messaging protocol incurs additional context switches, although the relative overhead is less as a percentage. In addition to the increased mean end-to-end latency, higher standard deviations are observed when LLFT is used. These results are as expected, given the additional services that LLFT provides.

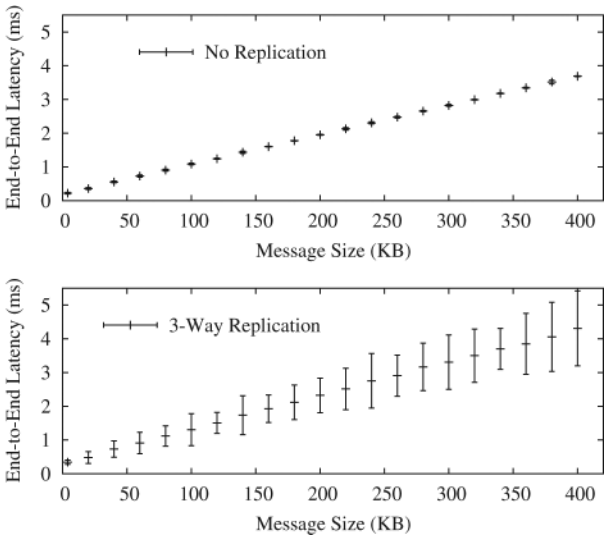

FIGURE 12. End-to-end latency vs. message size.

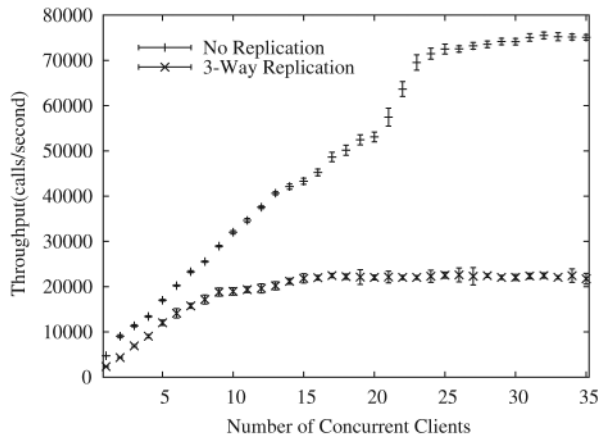

FIGURE 13. Throughput vs. number of concurrent clients.

We also measured the throughput, without replication using TCP and with three-way replication using LLFT, in the presence of various numbers of concurrent clients. Each client repeatedly issues 1 KB requests without any think time, and the server responds with 1 KB replies. The throughput results are summarized in Fig. 13. As is evident, although the mean

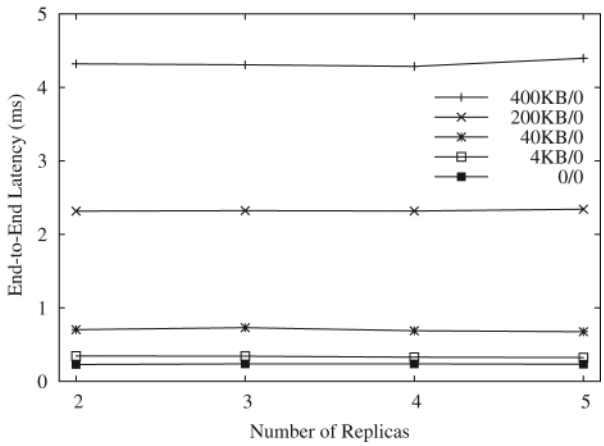

FIGURE 14. End-to-end latency vs. the number of replicas in a group.

throughput reduction with replication is moderate under light loads, it is more prominent under heavy loads.

We also characterized the fault scalability of the messaging protocol. As shown in Fig. 14, the performance for various request/reply message sizes does not degrade noticeably as the number of replicas is increased (so that larger numbers of concurrent faults can be tolerated). These results are as expected because LLFT is explicitly designed to allow the primary to deliver a message as soon as it is ordered within a connection without the need to communicate with the backups, thus minimizing the latency during normal operation. To avoid obscuring the comparison, error bars are not shown in this figure.

To demonstrate the benefits of using LLFT for latency-sensitive applications, we compared the end-to-end latency of LLFT with Totem [17]. We used the same test application for (1) no-replication, (2) three-way replication with LLFT and (3) three-way replication with Totem (one Totem instance runs on each node, and the client or server replica connects to the local Totem instance). As shown in Fig. 15, LLFT outperforms Totem by a large margin. The probability density function (PDF) for LLFT is much closer to that for the non-replicated application. The mean latency for Totem is about four times that for LLFT, and the standard deviation for Totem is more than four times that for LLFT.

The much higher mean end-to-end latency and standard deviation for Totem can be explained as follows:

(i) In Totem, a node must wait to receive the circulating token before it can multicast a message. This waiting time increases the end-to-end latency, and causes greater unpredictability, which leads to a higher standard deviation. LLFT does not suffer from this problem because the node can send its message immediately.

(ii) In Totem, the nodes form a logical ring and the token circulates around the ring. When a node acquires the token and has messages to send, an additional delay occurs, forcing the nodes downstream to wait longer to receive the token. Again, this results in a higher end-to-end latency and standard deviation.

6.2. Membership protocol

To evaluate the performance of the membership protocol during recovery after detection of a fault in the primary, we considered (1) the primary view change latency and (2) the recovery latency, i.e. the primary view change latency plus the virtual synchrony latency (which includes the time to communicate the application state). The failover latency is determined by the fault detection time and the recovery latency. In a system that does not incur lengthy communication delays, the first backup can detect a fault in the primary in $\sim$30 ms, based on the parameters used. In our experiments, the application state is small and the time to communicate it is not significant.

This evaluation is performed for crash faults because all faults (including crash faults, timing faults, and partitioning faults) are detected as timing faults by the fault detectors. Thus, they have similar performance characteristics for this evaluation.

Figure 16 summarizes the measurement results for the mean primary view change latency with error bars for the standard deviation. The measurement results were obtained with no clients running to highlight the primary view change latency. As the figure shows, the mean primary view change latency increases with the number of replicas. Interestingly, when the number of replicas is two (which industry regards as the typical case and which majority-based membership algorithms do not handle), the mean primary view change latency is < 0.05 ms, which is significantly less than the latency with more replicas, and much less than the latency for other membership protocols. In this case, when the primary crashes, only one replica remains. That replica can promote itself to be the new primary without waiting for acknowledgments from other replicas.

Figure 17 summarizes the measurement results for the recovery latency, i.e. the primary view change latency plus the virtual synchrony latency. The figure shows the measured mean recovery latency in the presence of various numbers of concurrent clients, for two-way and three-way replication. As expected, the mean recovery latency increases with the number of concurrent clients in both cases. If the availability requirement allows two-way replication (which is typical industrial practice), the recovery is faster by $\sim$0.2 ms. Again, to avoid obscuring the comparison, error bars are not shown in this figure.

6.3. Virtual determinizer framework

To evaluate the performance of the virtual determinizer framework, we injected non-deterministic operations into our benchmark application. For each run, we varied the number of non-deterministic operations per call, while keeping the request/reply message size fixed at 1 KB.

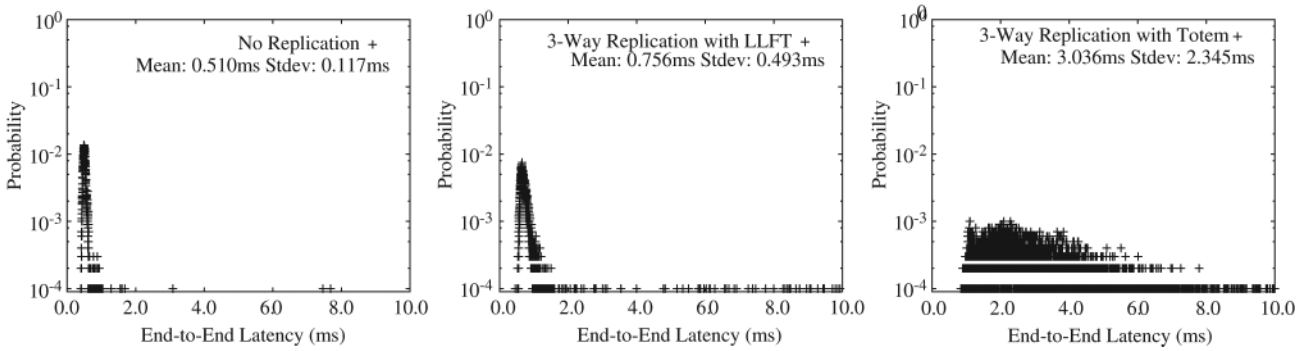

FIGURE 15. PDFs for the end-to-end latency for no replication with TCP, three-way replication with LLFT and three-way replication with Totem.

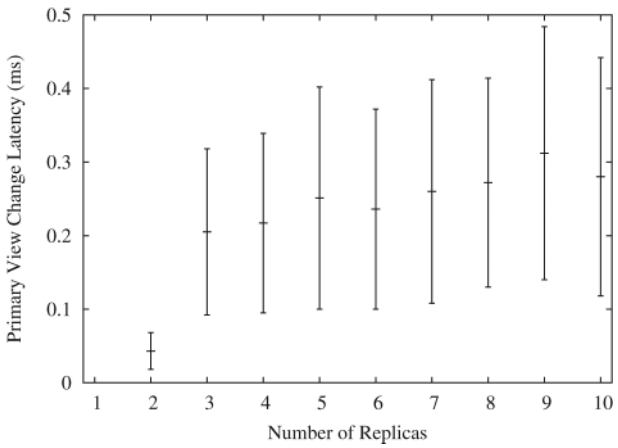

FIGURE 16. Primary view change latency.

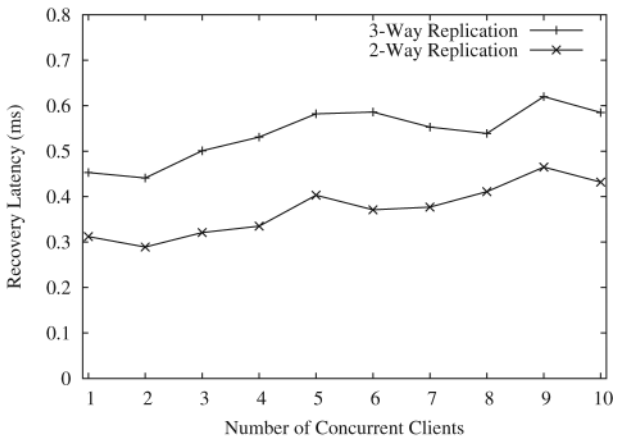

FIGURE 17. Recovery latency.

The measurement results, shown in Fig. 18, were obtained by introducing a clock-related non-deterministic operation (i.e. `gettimeofday()`) into the application. The figure shows the mean end-to-end latency with error bars corresponding to the

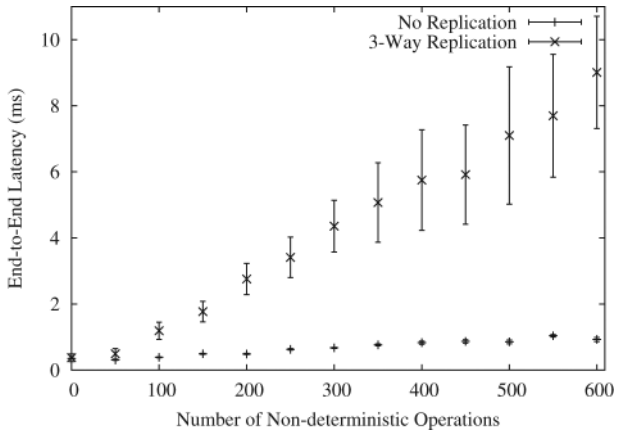

FIGURE 18. End-to-end latency vs. the number of non-deterministic operations.

standard deviation. Other kinds of non-deterministic operations produce similar profiles. In general, the mean end-to-end latency increases linearly as the number of non-deterministic operations per call increases. On average, each additional non-deterministic operation adds ∼0.008 ms overhead to the end-to-end latency. This overhead is primarily due to the piggybacking of ordering information.

7. RELATED WORK

The LLFT system provides fault tolerance transparently to both the applications and the operating system, like the TARGON/32 [23], TFT [24] and Hypervisor [25] systems. Those systems differ from LLFT in the way in which they achieve transparency. The TARGON/32 system uses a special bus design that ensures atomic transmission of a message sent by a primary to both a destination group and its own backups. The TFT system requires application object code editing. The Hypervisor system requires a hardware instruction counter. LLFT uses the more flexible library interpositioning technique.

The LLFT system uses a leader–follower replication strategy similar to that used in the viewstamped replication [6] and Delta-4 [7] systems. In the viewstamped replication system, the primary generates a new timestamp each time it needs to communicate information to the backups. Unlike LLFT, the viewstamped replication system is based on atomic transactions, combined with a view change algorithm. The Delta-4 system uses a separate message to transmit ordering information from the primary to the backups. The primary must wait until all of the backups have explicitly acknowledged the ordering information before it sends its next message, which can increase the response time at the client. In contrast, LLFT uses piggybacking and reflection mechanisms to reduce the end-to-end latency.

Atomic multicast protocols that deliver messages reliably and in total order, such as Isis [4], Amoeba [16], Totem [17], Newtop [18], Coyote [26] and Spread [27], have been used to maintain strong replica consistency in fault-tolerant distributed systems. However, those protocols introduce delays in either sending or delivering messages. The LLFT messaging protocol does not incur such delays because the primary decides on the order in which the operations are performed and the ordering information is reflected to its backups. The LCR total order broadcast protocol [28], which uses logical clocks and a ring topology, optimizes for high throughput in cluster environments rather than low latency as does LLFT. LCR is comparable to the Totem single-ring protocol [29], which likewise optimizes for high throughput rather than low latency. Some group communication systems, such as Horus [30], Arjuna [31] and Cactus [32], are presented as toolkits from which a high efficiency system can be constructed for a specific application. The effective use of such toolkits requires substantial skill. LLFT is designed to achieve comparable performance without customization and specialized skill.

A comprehensive survey of membership protocols and group communication systems, and of their formal specifications, is provided by Chockler *et al.* [33]. Schiper and Toueg [34] provide an elegant formalization of the dynamic membership problem that distinguishes between the problem of maintaining and agreeing on a set of members and the problem of determining which processes are working and should be members. Likewise, we distinguish between the group membership and the pool of potential members.

The Isis coordinator-cohort strategy [4] is somewhat like semi-passive replication in LLFT, but its membership protocol is quite different. As the Isis book states, 'each member has the same view of which process is the oldest in the group, so all agree implicitly on who the coordinator should be'. For this to work, first the members must reach agreement on the membership set and only then can they agree implicitly on the oldest member in that set. LLFT works the other way around. In LLFT, first the new primary is determined as the process that proposes to become the new primary and that has the highest precedence and, then, the new primary unilaterally determines the membership set. Thus, no agreement or consensus algorithm is required.

The Paxos algorithm [12] is a leader election algorithm for asynchronous distributed systems, that uses a two-phase commit strategy in which a majority of the members vote for the leader. Paxos assumes a known existing membership, and does not change the membership dynamically as members become faulty and recover. Paxos can achieve consensus in two rounds if communication is reliable and processes respond promptly. Like the partitionable membership of LLFT, Vertical Paxos [35] allows multiple configurations (views) to operate concurrently. Vertical Paxos is oriented toward primary-backup data replication, whereas LLFT is oriented toward primary-backup process replication.

Defago and Schiper [36] and Defago *et al.* [37] have investigated semi-passive replication along with a consensus algorithm. Their model admits non-deterministic operations, but not concurrent processing, with shared data, of requests from multiple clients. The primary server produces its results as a single action, including the reply to the client and the state update for the backups. In our model, multiple processes, possibly with multiple threads, can interact with each other. Requests from multiple clients can be processed concurrently and can access shared data. In their system, every server replica sends a reply to the client, whereas, in LLFT, the backups do not send replies to the client, which reduces the network traffic. Their system uses a rotating coordinator and consensus, whereas LLFT uses a leader-determined membership protocol without consensus. In [38], Saito and Shapiro provide a comprehensive survey of replication strategies.

Membership protocols for group communication systems, such as Totem [17] and Transis [21], employ fault detectors, based on timeouts, to reach distributed agreement on as large a membership as possible, devolving to smaller memberships, if necessary. Those membership protocols are relatively costly in the number of messages exchanged and in the delays incurred. To avoid such costs, the LLFT membership protocol does not involve distributed agreement but, rather, achieves consistent group membership by having the primary determine the membership and communicate it to the backups in the group. Moreover, LLFT does not attempt to form as large a membership as possible, like Transis and Totem do.

The LLFT virtual determinizer framework provides a generic algorithm and uniform data structures for capturing, transmitting and executing ordering information for non-deterministic operations to ensure replica consistency for such operations. The non-deterministic operations handled by LLFT overlap those considered in other systems such as Delta-4 [7], TARGON/32 [23], TFT [24] and Hypervisor [25]. To build a fault-tolerant Java virtual machine, Friedman and Kama [39] and Napper *et al.* [40] have extended the Hypervisor work to address non-determinism caused by multithreading. The Voltan environment [41] also provides deterministic replication for applications that have non-deterministic system calls.

Basile *et al.* [42], Jimenez-Peris and Arevalo [43] and Narasimhan *et al.* [44] have addressed the need to sanitize non-deterministic operations to achieve strong replica consistency for active replication, rather than for leader–follower (semi-active or semi-passive) replication. The LLFT mechanisms that are used to order mutex claims/releases are closely related to those of the Loose Synchronization Algorithm (LSA) and Preemptive Deterministic Scheduling Algorithm (PDS) of [42]. However, LSA does not address the strong replica consistency issues introduced by the `pthread_mutex_trylock()` library call, and PDS is suitable for only a specific threading model.

8. CONCLUSIONS AND FUTURE WORK

The LLFT system provides fault tolerance for distributed applications deployed over a local-area network, as in a single data center, cluster or cloud. Applications programmed using TCP socket APIs, or middleware such as Java RMI, can be replicated with strong replica consistency using LLFT, without any modifications to the applications. Performance measurements show that LLFT achieves low latency message delivery under normal conditions and low latency reconfiguration and recovery when a fault occurs. The genericity, application transparency and low latency of LLFT make it appropriate for a wide variety of distributed applications, particularly for latency-sensitive applications.

Future work includes sanitization of other sources of non-determinism (such as operating system signals and interrupts) and performance optimization. It also includes the development of more complex applications for LLFT (in particular, file systems and database systems), and the development of replication management tools.

FUNDING

This research was supported in part by NSF grant CNS-0821319, and by a CSISI grant from Cleveland State University (for the first author). An earlier abbreviated version of this paper appeared in [45].

REFERENCES

[1] Chandra, T.D. and Toueg, S. (1996) Unreliable failure detectors for reliable distributed systems. *J. ACM*, **43**, 225–267.

[2] Fischer, M.J., Lynch, N.A. and Paterson, M.S. (1985) Impossibility of distributed consensus with one faulty process. *J. ACM*, **32**, 374–382.

[3] Service Availability Forum. http://www.saforum.org. (accessed July 25, 2012).

[4] Birman, K.P. and van Rennesse, R. (1994) *Reliable Distributed Computing Using the Isis Toolkit*. IEEE Computer Society Press, Los Alamitos, CA.

[5] Moser, L.E., Amir, Y., Melliar-Smith, P.M. and Agarwal, D.A. (1994) Extended Virtual Synchrony. *Proc. 14th IEEE Int. Conf. Distributed Computing Systems*, Poznan, Poland, June 21–24, pp. 56–65. IEEE Computer Society Press, Los Alamitos, CA.

[6] Oki, B. and Liskov, B. (1988) Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems. *Proc. ACM Symp. Principles of Distributed Computing*, Toronto, Canada, August 15–17, pp. 8–17. ACM Publications, New York, NY.

[7] Powell, D. (1991) *Delta-4: A Generic Architecture for Dependable Distributed Computing*. Springer, Berlin, Germany.

[8] Gilbert, S. and Lynch, N.A. (2012) Perspectives on the CAP theorem. *IEEE Comput.*, **45**, 30–36.

[9] Asplund, M. and Nadjm-Terani, S. (2006) Post-Partition Reconciliation Protocols for Maintaining Consistency. *Proc. ACM Symp. Applied Computing*, Dijon, France, April 23–27, pp. 710–717. ACM Publications, New York, NY.

[10] Melliar-Smith, P.M. and Moser, L.E. (1998) Surviving network partitioning. *IEEE Comput.*, **31**, 62–69.

[11] Guerraoui, R., Hurfin, M., Mostefaoui, A., Oliveira, R., Raynal, M. and Schiper, A. (2000) Consensus in Asynchronous Distributed Systems: A Concise Guided Tour. *Advances in Distributed Systems*, pp. 33–47, Lecture Notes in Computer Science 1752. Springer, Berlin, Germany.

[12] Lamport, L. (1998) The part-time parliament. *ACM Trans. Comput. Syst.*, **16**, 133–169.

[13] Avizienis, A., Laprie, J. C., Randell, B. and Landwehr, C. (2004) Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Secur. Comput.*, **1**, 11–33.

[14] Cristian, F. and Fetzer, C. (1999) The timed asynchronous distributed system model. *IEEE Trans. Parallel Distrib. Syst.*, **10**, 603–618.

[15] Fetzer, C. (2003) Perfect failure detection in timed asynchronous systems. *IEEE Trans. Comput.*, **52**, 99–112.

[16] Kaashoek, M.F. and Tanenbaum, A.S. (1991) Group Communication in the Amoeba Distributed Operating System. *Proc. 11th IEEE Int. Conf. Distributed Computing Systems*, Arlington, TX, May 20–24, pp. 222–230. IEEE Computer Society Press, Los Alamitos, CA.

[17] Moser, L.E., Melliar-Smith, P.M., Agarwal, D.A., Budhia, R.K. and Lingley-Papadopoulos, C.A. (1996) Totem: a fault-tolerant multicast group communication system. *Commun. ACM*, **39**, 54–63.

[18] Ezhilchelvan, P.D., Macedo, R.A. and Shrivastava, S.K. (1995) Newtop: A Fault-Tolerant Group Communication Protocol. *Proc. 15th IEEE Int. Conf. Distributed Computing Systems*, Vancouver, BC, Canada, May 30–June 2, pp. 296–306. IEEE Computer Society Press, Los Alamitos, CA.

[19] Wensley, J.H., Lamport, L., Goldberg, J., Green, M.W., Levitt, K.N., Melliar-Smith, P.M., Shostak, R.E. and Weinstock, C.B. (1978) SIFT: design and analysis of a fault-tolerant computer for aircraft control. *Proc. IEEE*, **66**, 1240–1255.

[20] Terry, D.B., Theimer, M.M., Petersen, K., Demers, A.J., Spreitzer, M.J. and Hauser, C.H. (1995) Managing Update Conflicts in Bayou, A Weakly Connected Replicated Storage System. *Proc. 15th ACM Symp. Operating Systems Principles*, Copper Mountain Resort, CO, December 3–6, pp. 172–183. ACM Publications, New York, NY.

[21] Amir, Y., Dolev, D., Kramer, S. and Malkhi, D. (1992) Membership Algorithms for Multicast Communication Groups. *Proc. 6th Int. Workshop on Distributed Algorithms*, Haifa, Israel, November 2–4, pp. 292–312, Lecture Notes in Computer Science 647. Springer, Berlin, Germany.

[22] Dieter, W.R. and Lumpp, J.J.E. (2001) User-Level Checkpointing for LinuxThreads Programs. *Proc. 2001 USENIX Technical Conf.*, Boston, MA, June 25–30, pp. 81–92. USENIX, Berkeley, CA.

[23] Borg, A., Blau, W., Graetsch, W., Herrmann, F. and And, W. (1989) Fault tolerance under Unix. *ACM Trans. Comput. Syst.*, **7**, 1–24.

[24] Bressoud, T.C. (1998) TFT: A Software System for Application-Transparent Fault Tolerance. *Proc. 28th IEEE Int. Conf. Fault-Tolerant Computing*, Munich, Germany, June 23–25, pp. 128–137. IEEE Computer Society Press, Los Alamitos, CA.

[25] Bressoud, T.C. and Schneider, F.B. (1996) Hypervisor-based fault tolerance. *ACM Trans. Comput. Syst.*, **14**, 80–107.

[26] Bhatti, N.T., Hiltunen, M.A., Schlichting, R.D. and Chiu, W. (1998) Coyote: a system for constructing fine-grain configurable communication services. *ACM Trans. Comput. Syst.*, **16**, 321–366.

[27] Amir, Y., Danilov, C., Miskin-Amir, M., Schultz, J. and Stanton, J. (2004) The Spread Toolkit: Architecture and Performance. Tech. Report cnds-2004-1, John Hopkins Univ., Computer Science Dept.

[28] Guerraoui, R., Levy, R. and Pochon, B. (2010) Throughput optimal total order broadcast for cluster environments. *ACM Trans. Comput. Syst.*, **28**, 1–32.

[29] Amir, Y., Moser, L.E., Melliar-Smith, P.M., Agarwal, D.A. and Ciarfella, P. (1995) The Totem single-ring ordering and membership protocol. *ACM Trans. Comput. Syst.*, **13**, 311–342.

[30] van Renesse, R., Birman, K.P. and Maffeis, S. (1996) Horus: a flexible group communication system. *Commun. ACM*, **39**, 76–83.

[31] Parrington, G.D., Shrivastava, S.K., Wheater, S.M. and Little, M.C. (1995) The design and implementation of Arjuna. *USENIX Comput. Syst. J.*, **8**, 255–308.

[32] Hiltunen, M.A. and Schlichting, R.D. (2000) The Cactus Approach to Building Configurable Middleware Services. *Proc. Workshop on Dependable System Middleware and Group Communication*, Nuremberg, Germany, October 16–18. IEEE Computer Society Press, Los Alamitos, CA.

[33] Chockler, G.V., Keidar, I. and Vitenberg, R. (2001) Group communication specifications: a comprehensive study. *ACM Comput. Surv.*, **33**, 427–469.

[34] Schiper, A. and Toueg, S. (2006) From set membership to group membership: a separation of concerns. *IEEE Trans. Dependable Secur. Comput.*, **3**, 2–12.

[35] Lamport, L., Malkhi, D. and Zhou, L. (2009) Vertical Paxos and primary-backup replication. Microsoft, Tech. Report MSR-TR-2009-63.

[36] Defago, X. and Schiper, A. (2004) Semi-passive replication and lazy consensus. *J. Parallel Distrib.Comput.*, **64**, 1380–1398.

[37] Defago, X., Schiper, A. and Sergent, N. (1998) Semi-Passive Replication. *Proc. 17th IEEE Symp. Reliable Distributed Systems*, West Lafayette, IN, October 20–22, pp. 43–50. IEEE Computer Society Press, Los Alamitos, CA.

[38] Saito, Y. and Shapiro, M. (2005) Optimistic replication. *ACM Comput. Surv.*, **37**, 42–81.

[39] Friedman, R. and Kama, A. (2003) Transparent Fault-Tolerant Java Virtual Machine. *Proc. 22nd IEEE Symp. Reliable Distributed Systems*, Florence, Italy, October 6–8, pp. 319–328. IEEE Computer Society Press, Los Alamitos, CA.

[40] Napper, J., Alvisi L. and Vin, H. (2003) A Fault-Tolerant Java Virtual Machine. *Proc. IEEE Int. Conf. Dependable Systems and Networks*, San Francisco, CA, June 22–25, pp. 425–434. IEEE Computer Society Press, Los Alamitos, CA.

[41] Black, D., Low, C. and Shrivastava, S.K. (1998) The Voltan application programming environment for fail-silent processes. *Distrib. Syst. Eng.*, **5**, 66–77.

[42] Basile, C., Kalbarczyk, Z. and Iyer, R. (2006) Active replication of multithreaded applications. *IEEE Trans. Parallel Distrib. Syst.*, **17**, 448–465.

[43] Jimenez-Peris, R. and Arevalo, S. (2000) Deterministic Scheduling for Transactional Multithreaded Replicas. *Proc. 19th IEEE Symp. Reliable Distributed Systems*, Nuremberg, Germany, October 16–18, pp. 164–173. IEEE Computer Society Press, Los Alamitos, CA.

[44] Narasimhan, P., Moser, L.E. and Melliar-Smith, P.M. (1999) Enforcing Determinism for the Consistent Replication of Multithreaded CORBA Applications. *Proc. 18th IEEE Symp. Reliable Distributed Systems*, Lausanne, Switzerland, October 19–22, pp. 263–273. IEEE Computer Society Press, Los Alamitos, CA.

[45] Zhao, W., Melliar-Smith, P.M. and Moser, L.E. (2010) Fault Tolerance Middleware for Cloud Computing. *Proc. IEEE 3rd Int. Conf. Cloud Computing*, Miami, FL, July 5–10, pp. 67–74. IEEE Computer Society Press, Los Alamitos, CA.

APPENDIX

The proofs of correctness for LLFT, based on the model and the safety and liveness properties given in Section 2.6, are provided below.

THEOREM A.1 (Safety). *At most one infinite sequence of consecutive primary views exists. Each of those consecutive primary views has a unique consecutive primary view number and a single primary replica.*

Proof. Assume that the primary R_1 became faulty in view V_i, and that the backup R_2 with precedence p_2 and the backup R_3 with precedence p_3, where $p_2 < p_3$, each propose a new primary view. The two primary views have the same primary view number but different new primaries (either R_2 or R_3). Consider the following two cases.

Case 1. There is a replica R that is a member of both proposed new primary views. According to the rules of the LLFT membership protocol, if R first acknowledges R_3's `ProposePrimary` message, then R does not acknowledge R_2's `ProposePrimary` message because $p_2 < p_3$. Thus, by the eventual reliable communication assumption, when R_2

receives R's acknowledgment to R_3, R_2 abandons its attempt to form that new membership, resets its state and applies to rejoin the group. On the other hand, if R first acknowledges R_2's `ProposePrimary` message and subsequently receives R_3's `ProposePrimary` message, then R acknowledges R_3's `ProposePrimary` message because $p_2 < p_3$. When R_2 receives R's acknowledgment to R_3, R_2 abandons its attempt to form that new membership, resets its state and applies to rejoin the group.

Case 2. There is no replica that is a member of both proposed new primary views (because neither R_2 nor R_3 received messages from any replica in the other's proposed membership and, thus, R_2 and R_3 both regard the other's replicas as faulty). Because of the eventual reliable communication assumption and because a message containing a higher precedence than the precedence of the primary of the primary view is sent and retransmitted to all members of the group, every replica R in R_2's membership eventually receives a message from a replica in R_3's membership. Because $p_2 < p_3$, R then realizes that R_2's membership has been superseded by R_3's membership and, thus, R abandons its current state and applies for readmission to R_3's membership. Thus, any side branch is pruned.

Note that there cannot be two replicas with the same precedence because (1) if a primary admits multiple replicas to the membership, it assigns different precedences to each of the replicas; (2) each such precedence is qualified by the precedence of the primary that admitted the replica to the membership; and (3) the primary precedences form a chain back to the unique initial primary replica for the group.

In the theorems and proofs below, operations refer to both computation and communication operations. □

THEOREM A.2 (Safety). *At most one infinite sequence of operations in an infinite sequence of consecutive primary views exists.*

Proof. By Theorem A.1, there exists at most one infinite sequence of consecutive primary views. Each of those primary views has a unique primary view number and a single primary replica. Moreover, each of those primary views has an associated sequence of operations determined by the primary of that view. The sequence of operations in an infinite sequence of consecutive primary views is the concatenation of the sequences of operations for the primary views in the order of their primary view numbers. □

LEMMA A.1. *In semi-active replication, if a backup replica R_2 is admitted to the membership of view V_i by primary replica R_1 after the start of V_i and R_2 subsequently becomes faulty in V_i, then the sequence of operations of R_2 in V_i is a consecutive subsequence of the sequence of operations of R_1 in V_i.*

Proof. For primary replica R_1 of view V_i, the sequence of ordering information is determined by the sequence of operations of R_1. When the backup R_2 is admitted to the membership of view V_i by R_1 after the start of view V_i, it receives a `State` message from R_1, which establishes a synchronization point. After that point, the sequence of operations performed by R_2 is determined by the sequence of ordering information provided by R_1, until R_2 becomes faulty. Thus, the sequence of operations of R_2 in V_i is a consecutive subsequence of the sequence of operations of R_1 in V_i. □

LEMMA A.2. *In semi-active replication, if replica R_2 is admitted to the membership of view V_i by primary replica R_1 after the start of V_i and R_2 is not faulty in V_i, then the sequence of operations of R_2 in V_i is a suffix of the sequence of operations of R_1 in V_i.*

Proof. For primary replica R_1 of view V_i, the sequence of ordering information is determined by the sequence of operations of R_1. When R_2 is admitted to the membership of view V_i by R_1 after the start of V_i, it receives a `State` message from R_1, which establishes a synchronization point. After that point, the sequence of operations of R_2 is determined by the sequence of ordering information provided by R_1 in V_i. Moreover, because R_2 is not faulty in V_i, it participates in the virtual synchrony at the start of V_{i+1}. Thus, R_2's sequence of operations in V_i is a suffix of the sequence of operations of R_1 in V_i. □

LEMMA A.3. *In semi-active replication, if replica R_2 is an initial member of view V_i with primary replica R_1 and R_2 subsequently becomes faulty in V_i, then the sequence of operations of R_2 in V_i is a prefix of the sequence of operations of R_1 in V_i.*

Proof. For primary replica R_1 of view V_i, the sequence of ordering information is determined by the sequence of operations of R_1. Because R_2 is an initial member of view V_i, it participates in the virtual synchrony at the start of V_i. After that point, the sequence of operations of R_2 is determined by the sequence of ordering information provided by R_1 in V_i, until R_2 becomes faulty. Thus, R_2's sequence of operations in V_i is a prefix of the sequence of operations of R_1 in V_i. □

LEMMA A.4. *In semi-active replication, if replicas R_2 and R_3 are members of the same memberships of views V_i, V_{i+1} and V_{i+2}, then the sequence of operations of R_2 in V_{i+1} is the same as the sequence of operations of R_3 in V_{i+1}.*

Proof. Because R_2 and R_3 are members of the same memberships of views V_i and V_{i+1}, both of them participate in the virtual synchrony between V_i and V_{i+1}, determined by primary replica R_1 of V_{i+1}. Because both R_2 and R_3 are members of the same memberships of views V_{i+1} and V_{i+2}, neither of them becomes faulty in V_{i+1} and both of them

participate in the virtual synchrony between V_{i+1} and V_{i+2}. Consequently, both R_2 and R_3 perform the same sequence of operations in V_{i+1}, which is the same as the sequence of operations performed by R_1, determined by the sequence of ordering information provided by R_1. □

LEMMA A.5. *In semi-active replication, if replicas R_2 and R_3 are members of the same memberships of views V_i and V_k, then the sequence of operations of R_2 in V_j is the same as the sequence of operations of R_3 in V_j, where $i < j < k$.*

Proof. Because R_2 and R_3 are members of the same memberships of views V_i and V_k, they are both members of the same memberships of views V_j for all $j, i < j < k$, because if they are removed from a membership and apply for readmission, they are admitted as new members. The proof follows from Lemma A.4 by induction. □

THEOREM A.3 (Safety). *In semi-active* (*semi-passive*) *replication, for a member in a view of the infinite sequence of consecutive primary views, the sequence of operations* (*states*) *of that member is a consecutive subsequence of the infinite sequence of operations* (*states*) *of the group.*

Proof. Based on the above lemmas, we provide the proof for semi-active replication. The proof for semi-passive replication is similar. We consider three cases:

Case 1: Replica R is admitted to a membership in view V_i and becomes faulty in the same view V_i. By Lemma A.1, the sequence of operations of R in V_i is a consecutive subsequence of the sequence of operations of primary replica R_1 in V_i and, thus, of the infinite sequence of operations of the group.

Case 2: Replica R is admitted to a membership in view V_i and becomes faulty in view V_{i+1}. By Lemma A.2, the sequence of operations of R in V_i is a suffix of the sequence of operations of primary replica R_1 in V_i. By Lemma A.3, the sequence of operations of R in V_{i+1} is a prefix of the sequence of operations of primary replica R_1' in V_{i+1}. Thus, the sequence of operations of R is the concatenation of the suffix for view V_i and the prefix for view V_{i+1}. Thus, the sequence of operations of R is a consecutive subsequence of the infinite sequence of operations of the group.

Case 3: Replica R is admitted to the membership in view V_i and becomes faulty in view V_k, where $k > i+1$. By Lemma A.2, the sequence of operations of R in V_i is a suffix of the sequence of operations of primary replica R_1 in V_i. By Lemma A.3, the sequence of operations of R in V_k is a prefix of the sequence of operations of primary replica R_1' in V_k. By Lemma A.5, the sequence of operations of R in V_j is the same as the sequence of operations of primary replica R_1'' in V_j, where $i < j < k$. Thus, the sequence of operations of R is the concatenation of the suffix for view V_i, the sequences for the views V_j, where $i < j < k$, and the prefix for view V_k. Thus, the sequence of operations of R is a consecutive subsequence of the infinite sequence of operations of the group.

In the proofs above, which apply to semi-active replication, we consider the sequence of *operations* performed at the primary and at the backups. To address semi-passive replication, we need to consider the sequence of *states* at the primary and the backups, because for semi-passive replication, the backups perform no operations. □

THEOREM A.4 (Liveness). *At least one infinite sequence of consecutive primary views with consecutive primary view numbers exists.*

Proof. By the sufficient replication assumption (i.e. each group contains enough replicas such that in each primary view there exists at least one replica that does not become faulty), if the primary becomes faulty in a view V_i, then there exists a replica R in V_i that can assume the role of the primary in view V_{i+1}. The proof follows by induction. □

THEOREM A.5 (Liveness). *At least one infinite sequence of operations in an infinite sequence of consecutive primary views exists.*

Proof. There exists at least one operation (the communication of the `State` message) in each primary view. The proof now follows from Theorem A.4. □