2010 IEEE 3rd International Conference on Cloud Computing

Fault Tolerance Middleware for Cloud Computing

Wenbing Zhao
Department of Electrical and Computer Engineering
Cleveland State University
Cleveland, OH 44115
wenbing@ieee.org

P. M. Melliar-Smith and L. E. Moser
Department of Electrical and Computer Engineering
University of California, Santa Barbara
Santa Barbara, CA 93106
pmms@ece.ucsb.edu, moser@ece.ucsb.edu

***Abstract*—The Low Latency Fault Tolerance (LLFT) middleware provides fault tolerance for distributed applications deployed within a cloud computing or data center environment, using the leader/follower replication approach. The LLFT middleware consists of a Low Latency Messaging Protocol, a Leader-Determined Membership Protocol, and a Virtual Determinizer Framework. The Messaging Protocol provides a reliable, totally ordered message delivery service by employing a direct group-to-group multicast where the ordering is determined by the primary replica in the group. The Membership Protocol provides a fast reconfiguration and recovery service when a replica becomes faulty and when a replica joins or leaves a group. The Virtual Determinizer Framework captures ordering information at the primary replica and enforces the same ordering at the backup replicas for major sources of non-determinism. The LLFT middleware maintains strong replica consistency, offers application transparency, and achieves low end-to-end latency.**

***Keywords*-cloud computing; fault tolerance; replication**

I. Introduction

Cloud computing aims to provide reliable services within data centers that contain servers, storage and networks. The services are delivered to the users transparently without their need to know the details of the underlying software and hardware. One of the challenges of cloud computing is to ensure that the applications run without a hiatus in the services they provide to the users. The Low Latency Fault Tolerance (LLFT) middleware provides fault tolerance for distributed applications deployed within a cloud computing or data center environment, as a service offered by the owners of the cloud.

The LLFT middleware replicates the processes of the applications, using the leader/follower replication approach [1]. The replicas of a process constitute a process group, and the process groups that provide a service to the users constitute a service group, as shown in Figure 1. Within a process group, one replica is designated as the primary, and the other replicas are the backups. The primary in a process group multicasts messages to a destination process group over a virtual connection. The primary in the destination process group orders the messages, performs the operations, and produces ordering information for non-deterministic operations, which it supplies to the backups.

The LLFT middleware maintains strong replica consistency of the states of the replicas, both in fault-free operation and in the event of a fault. If a fault occurs, the LLFT reconfiguration/recovery mechanisms ensure that a backup has, or can obtain, the messages and the ordering information it needs to reproduce the actions of the primary. They transfer the state from an existing replica to a new replica and synchronize the operation of the new replica with the existing replicas, to maintain virtual synchrony [2].

The novel contributions of the LLFT middleware include the Low Latency Messaging Protocol, the Leader-Determined Membership Protocol, and the Virtual Determinizer Framework.

The Low Latency Messaging Protocol provides a reliable, totally ordered multicast service by communicating message ordering information from the primary in a group to the backups in the group. It achieves low latency (minimal overhead in the response time seen at a client), by having the primary make the decisions on the order in which operations are performed and reflecting the ordering information to the backups. It involves fewer messages than prior fault-tolerant systems by multicasting messages, piggybacking information, suppressing messages at the backups, *etc*.

The Leader-Determined Membership Protocol ensures that the members of a group have a consistent view of the membership set and of the primary member of 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 ranks and precedences of the backups and by avoiding the need for a multi-round consensus algorithm.

The Virtual Determinizer Framework renders the applications virtually deterministic by recording the order and results of each non-deterministic operation carried out by the primary, and by guaranteeing that the backups obtain the same results in the same order as the primary.

II. Basic Concepts

A. Fault Model

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

978-0-7695-4130-3/10 $26.00 © 2010 IEEE
DOI 10.1109/CLOUD.2010.26

IEEE computer society

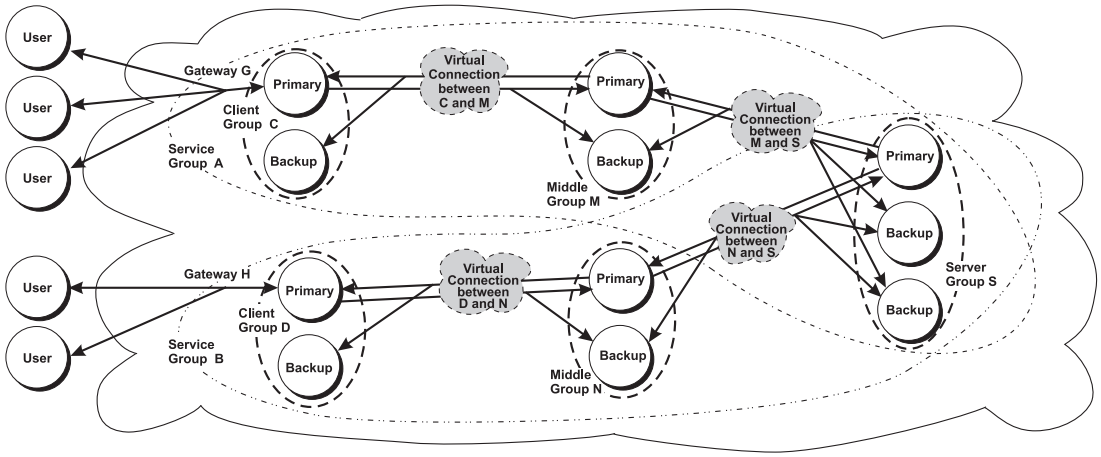

Figure 1. Service groups consisting of multiple process groups interacting over virtual connections within a cloud.

- *Crash fault* - A process or processor does not produce any further results.
- *Timing fault* - A process or processor does not produce a result within a given time constraint.

The LLFT middleware does not handle Byzantine faults.

B. Replication Types

The LLFT middleware supports two types of leader/follower replication:

- *Semi-active replication* - The primary orders the messages it receives, performs the corresponding operations, and 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.
- *Semi-passive replication* - The primary orders the messages it receives, performs the corresponding operations, and provides ordering information for non-deterministic operations to the backups. In addition, the primary communicates state updates to the backups, as well as file and database updates. A backup receives and logs incoming messages, and updates its state, but does not perform the operations and does not produce outgoing messages.

Semi-passive replication uses fewer processing resources than semi-active replication, but it incurs greater latency for reconfiguration and recovery, if the primary fails.

C. Service Groups and Process Groups

A *service group* is a set of interacting process groups that collectively provide a service to the user. A *process group (virtual process)* is a set of replicas of a process. Typically, different replicas in a process group run on different processors. One of the members in the process group is the *primary*, and the other members are the *backups*. See Figure 1.

Each process group has a *group membership* that comprises the replicas of the process. A membership change that introduces a new primary constitutes a new *primary view*, which has a *primary view sequence number*. Each member of a process group must know the primary of its group. There is no need for the members of a sending process group to know which member of a destination process group is the primary.

D. Virtual Connections

The LLFT middleware 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 connection between two endpoints, where each endpoint is a process group and over which messages are communicated between the two group endpoints. A virtual connection is a full-duplex, many-to-many communication channel between the two endpoints. A sender uses UDP multicast 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 process group listen on the same virtual port, and members of different process groups listen on different virtual ports. The process groups need to know the virtual ports (group ids) of the process groups with which they are interacting, just as with TCP. Each service group has a port, by which the user accesses the service provided by the service group through a gateway into the cloud.

Typically, a process group is an endpoint of more than one virtual connection, and the members of a process group are participants in more than one virtual connection, as shown in Figure 1.

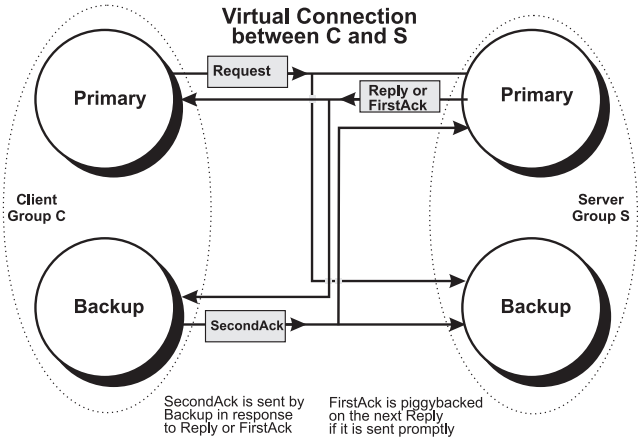

Figure 2. Message exchange between client and server groups.

III. LLFT Messaging Protocol

The LLFT Messaging Protocol provides the following services for the application messages:

- *Reliable delivery* - All of the members in a group receive each message that is multicast to the group on a connection.
- *Total ordering* - All of the members in a group deliver the messages to the application in the same sequence.

The LLFT Messaging Protocol converts the unreliable 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 IP unicast into a reliable, totally ordered message delivery service between two individual endpoints. If a fault occurs, the LLFT Messaging Protocol delivers messages to the application while maintaining virtual synchrony [2]. The LLFT Messaging Protocol incorporates flow control mechanisms similar to those of TCP.

A. Reliable Delivery Service

The types of messages used by the LLFT Messaging Protocol are discussed below for a Request from a client group C to a server group S, as shown in Figure 2. The same considerations apply for a Reply from the server group S to the client group C.

The primary in group C multicasts messages originated by the application to a destination group S over a virtual connection. A backup in group C creates and logs (but does not multicast) messages originated by the application. Restriction of the backup in this way reduces the amount of network traffic.

When the primary in group C first multicasts an application message to group S on a connection, it stores the message in a `sent list` for that connection. The primary retransmits a message in the `sent list` if it does not receive an acknowledgment for the message sufficiently promptly (as determined by a timeout). This timeout (and also the timeouts below) depend on the probability of message loss and on the processing and communication latency in the cloud computing enviroment.

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 the primary in group C on that connection. If the primary in group S does not have a message to multicast on the connection sufficiently promptly (as determined by a timeout), it multicasts a `FirstAck` message containing an `ack` for the last message it received without a gap.

The primary (and a backup) in group S adds the application messages that it receives on the connection to a `received list` for the connection. The replica also adds an entry corresponding to a received message to an `ack list`. If the replica detects a gap in the sequence numbers, it creates placeholders for missing messages, and adds corresponding entries to a `nack list`. If the replica receives a retransmitted message, and has a placeholder for the message, it replaces the placeholder with the message. Otherwise, it discards the message.

A backup in group C acknowledges a `FirstAck` message, that it receives, with a `SecondAck` message. The backup sends a `SecondAck` message in response to receiving a `FirstAck` message only if the backup application has already generated the message that the `FirstAck` message acknowledges. The primary in group S stops retransmitting a `FirstAck` message on receiving the corresponding `SecondAck` message.

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.

If the primary in group C does not receive a `FirstAck` message acknowledging the last message it sent, the primary (and a backup) in group S lags too far behind, perhaps because the primary in group S is handling requests from many client groups. In this case, the primary in group C invokes inter-group flow control mechanisms and slows down, so that the primary (and the backups) in group S can catch up. If the primary in group C receives many `FirstAck` messages acknowledging the last message it sent, the backups in group C did not send `SecondAck` messages and lag too far behind. In this case, the primary in group C invokes the intra-group coordination mechanisms and slows down, so that the backups in group C can catch up.

If the primary or a backup in group S determines that it has not received a message from the primary in group C on a connection, it multicasts a `Nack` message on the connection.

The primary and each backup in group S periodically exchange `Heartbeat` messages on the intra-group con-

nection, so that the primary knows that the backup has not failed (and vice versa).

B. Totally Ordered Delivery Service

The primary in a group communicates ordering information to the backups in its group, so that they can reproduce the actions of the primary and maintain strong replica consistency, if the primary fails.

The primary in group C piggybacks, on each message that it originates and sends on a connection, the ordering information for the messages that it has sent on the connection and received on the connection since the last message it sent on the connection (along with ordering information for other types of operations, as described in Section V). A backup in group C does not receive the ordering information directly from the primary in group C. Thus, the primary in group S reflects back the ordering information to group C in the next message it multicasts to group C. The primary in group C includes the ordering information in each message it sends until it receives that information reflected back to it. Similarly, the primary in group C reflects back to group S the ordering information that it receives in messages from the primary in group S.

IV. LLFT Membership Protocol

The LLFT Membership Protocol ensures that the members of a process group have a consistent view of the membership set and of the primary in the group. By making a deterministic choice of the primary, the LLFT Membership Protocol is much faster than a multi-round consensus protocol (which is particularly important when the primary fails). The primary determines the addition (removal) of the backups to (from) the group, based on their precedences and ranks.

The *precedence* of a member of a group is determined by the order in which the member joins the group. If a member fails and subsequently restarts, it is considered a new member, and is assigned a new precedence. The precedence numbers increase monotonically so that, in the history of a group membership, no two members have the same precedence and a member that joins later has a higher precedence. When a primary adds a new backup to the membership, it assigns the next precedence in sequence to that backup. However, the precedences of the members are not necessarily consecutive, because a member might have been removed from the group. The precedences of the members determine the order of succession to become the new primary, if the current primary fails.

The *rank* of the primary member of a group is 1, and the ranks of the backups are $2, 3, \ldots$ When a proposed new primary assigns ranks to the backups of a new membership or when a primary adds a new backup to the membership, it assigns those ranks in the order of their precedences. The ranks determine the timeouts for detection of failure of the primary or a backup.

Only membership changes that correspond to a change of the primary constitute a new view, which we refer to as a *primary view change*. Each such new primary view has a *primary view sequence number*. When the primary view changes, the proposed new primary resets the message sequence number to one and adjusts the members' ranks.

It is important for the backups to change the primary view at exactly the same virtual synchrony point as the primary. To this end, the new primary produces an ordering information entry for the primary view change and multicasts that entry to the backups, just like the other ordering information. A backup changes to the new primary view when it has performed all of the operations that were ordered before the virtual synchrony point.

A. Change of the Primary

The change of the primary replica in a group is handled in two phases, as described below.

1) Determining the New Membership: In the first phase, the new primary is determined. The new primary determines which backups are included in the membership.

If a backup detects that the primary is faulty because it has not received a `Heartbeat` message from the primary within a given time period, it multicasts a `ProposePrimary` message on the intra-group connection, denouncing the old primary and appointing itself as the new primary. The backup includes, in the `ProposePrimary` message, the group identifier, the old primary view number, its precedence, and the proposed new membership. The backup does not include, in the proposed new membership, any member that has a lower precedence than itself in the old membership.

A backup in the proposed new membership acknowledges the `ProposePrimary` message, unless it has also generated a `ProposePrimary` message. If two backups propose a new membership, only one of them can receive acknowledgments from all of the members in the membership it proposes.

When the proposed new primary has received acknowledgments from all members of the new primary view with itself as the new primary for that view, it concludes the first phase and proceeds to the second phase.

2) Recovering from the Membership Change: In the second 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 new primary collects information for the virtual synchrony point by sending a `NewPrimaryView` message on each inter-group connection. The `NewPrimaryView` message contains the most recent ordering information known to the new primary for that connection.

The acknowledgments for the `NewPrimaryView` message contain ordering information held by the primary of the remote group for the connection. The new primary might

determine that it has missed one or more messages sent by members of other groups through the ordering information contained in the acknowledgments it receives. The new primary sends `Nacks` for the missing messages until it receives all of them.

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 number and adjusting the members' ranks, and by generating an ordering information entry for the start of a new primary view. The backups switch to the new primary view when they receive and process the ordering information.

B. Change of a Backup

The change of a backup is either the addition of a backup to, or the removal of a backup from, a group.

1) Addition of a Backup: The backup first multicasts a `ProposeBackup` message on the intra-group connection. The primary assigns the rank and the precedence to the new backup and then multicasts, on the intra-group connection, an `AcceptBackup` message containing the new membership. A backup that receives an `AcceptBackup` message accepts the new membership and responds with an acknowledgment, if it is included in the new membership.

The new backup begins to log messages as soon as it starts up. The primary checkpoints its state once the new membership has been acknowledged by all of the backups in the group. Then, the state of the backup is set and the messages from the log are replayed.

If the primary becomes faulty before the new backup has received the checkpoint, one of the other backups proposes to become the new primary.

2) Removal of a Backup: The primary modifies the ranks of the backups in the group and then multicasts, on the intra-group connection, a `RemoveBackup` message, containing the new membership. When a backup receives a `RemoveBackup` message that includes itself in the membership, the backup accepts the new membership and responds with an acknowledgment. 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 to be readmitted.

V. LLFT Virtual Determinizer Framework

Applications deployed in a cloud computing environment typically involve various sources of non-determinism. 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 introduces a novel generic algorithm for sanitizing the sources of non-determinism. We have instantiated the generic algorithm for the following types of non-deterministic operations:

- Multi-threading
- Socket communication
- Time-related operations.

Due to space constraints, we discuss only multi-threading below. Other sources of non-determinism, such as operating system signals and interrupts, are not yet handled.

A. Threading Model

The state of an application process is determined by data that are shared among different threads, and by thread-specific data managed and changed by each thread locally.

Each thread within a process has a unique thread identifier. A data item that is shared by multiple threads is protected by a mutex. 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 a process group.

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

B. Generic Algorithm

The generic algorithm records the ordering information and 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. For each non-deterministic operation, the generic algorithm records the following information:

- *Thread identifier* - The identifier of the thread that is carrying out the operation
- *Operation identifier* - An identifier that represents one or more data items that might change during the operation or on completion of the operation
- *Operation count* - The number of operations carried out by a thread for a given operation identifier
- *Operation metadata* - The data returned from the system/library call. This metadata includes the `out` parameters (if any), the return value of the call, and the error code (if necessary).

At the primary, the algorithm maintains a queue, the `OrderInfo` queue of 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 is the `N`th time in its execution sequence that thread `T` has executed a non-deterministic call of interest. The `OrderInfo` queue spans different threads and different operations.

At the primary, the algorithm appends an entry `(T, O, N, D)` to the `OrderInfo` queue on return of the operation

O. The entries are transmitted to the backups reliably and in FIFO order, using the piggybacking mechanism of the LLFT Messaging Protocol.

At a backup, for each operation O, the algorithm maintains an O.OrderInfo queue of three-tuples (T, N, D), in the order in which the primary created them. After an entry is appended to the queue, the algorithm awakens an application thread if it is blocked.

At a backup, when thread T tries to execute the 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. It resumes a thread T that was suspended in the order in which (T, N, D) occurs in the O.OrderInfo queue, rather than in the order in which the thread was suspended or in 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. The algorithm requires the ordering of all related operations, such as claims and releases of mutexes.

C. Consistent Multi-Threading Service

The Consistent Multi-Threading Service (CMTS) is a specific instantiation of the generic algorithm, with a mutex *M* being used as the operation identifier. For the normal mutex claim call (pthread_ mutex_lock() library call), the metadata can be empty if the call is successful. However, if the normal mutex claim call returns an error code and for the nonblocking mutex claim call (pthread_mutex_trylock() library call), the mechanisms store the return value as the metadata in the recorded ordering information at the primary.

At a backup, when it is time to process a mutex ordering information entry, the CMTS mechanisms examine the metadata. If the metadata contains an error code, they return control to the calling thread with an identical error status, without performing the call. Otherwise, they delegate 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 owned the mutex releases it.

The Consistent Multi-Threading Service allows concurrency of threads that do not simultaneously acquire the same mutex, while maintaining strong replica consistency. Thus, it achieves the maximum degree of concurrency possible, while maintaining strong replica consistency.

VI. Implementation and Performance

The LLFT middleware is 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 the runtime environment. The application state is checkpointed and restored using facilities provided by a memory-mapped checkpoint library derived from [3]. The implementation is compiled into a shared library, which is inserted into the application address space at startup time using the LD_PRELOAD facility of the operating system. The LLFT middleware is transparent to the application, and does not require recompilation or relinking of the application program.

The experimental testbed consists of 14 HP blade servers, each equipped with two 2 GHz Intel Xeon processors, running the Linux Ubuntu 9.04 operating system, on a 1 Gbps Ethernet. A two-tier application that resembles a networked file system client/server was used to benchmark the LLFT implementation. The experiments involved three types of workloads: 0/0, x/0, and 0/x, where 0 denotes a small request or reply and x denotes a variable request or reply. The 0/0 workload characterizes operations on file metadata, the x/0 workload characterizes write operations in networked file systems, and the 0/x workload characterizes read operations.

The end-to-end latency measurement results for semi-active replication are shown in Figure 3. The graph on the left shows the dependency of the end-to-end latency on message size (requests or replies), with and without replication. The x/0 and 0/x workloads show no observable difference in end-to-end latency for the same x value. As can be seen in the figure, the LLFT Messaging Protocol incurs very moderate overhead, ranging from about 15% overhead for large messages to about 55% overhead for small messages. During normal operation, the overhead is primarily due to the piggybacking of ordering information. For large messages, which require fragmentation in user space, additional context switches are incurred.

Because of its design, LLFT offers excellent fault scalability, *i.e.*, the performance does not degrade as the replication degree is increased (so that the system can tolerate larger numbers of concurrent faults), as shown in the graph on the right of Figure 3.

In addition to the above experiments, we investigated the impact of system load on the end-to-end latency of LLFT. The load was introduced in the form of multiple concurrent clients issuing results to the server continuously without any think time. Due to the limited number of nodes in our testbed, we ran up to 10 concurrent clients and observed the end-to-end latency at each client. The end-to-end latency is insensitive to the load and remains identical to the value for the single-client case.

To study the performance of LLFT during fault recovery, we considered three scenarios: (1) recovery from primary failure, (2) recovery from backup failure, and (3) addition of a (new or restarted) replica. Because the LLFT Membership Protocol is very efficient, the recovery time from primary failure (scenario 1) is virtually entirely dependent on the

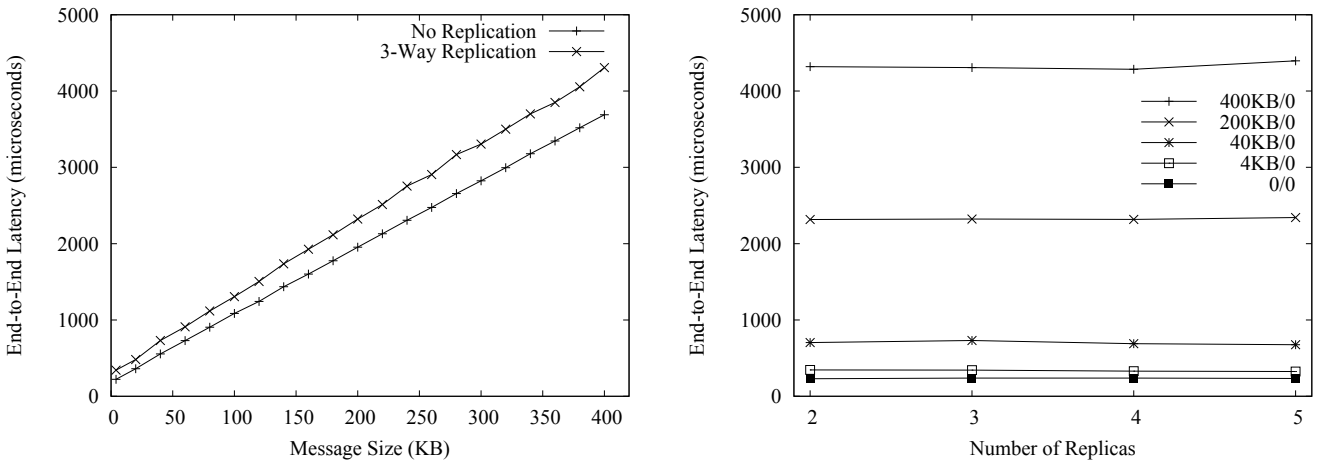

Figure 3. On the left, end-to-end latency with respect to various message sizes, with and without replication. On the right, end-to-end latency with respect to various replication degrees.

fault detection time, which ranges from 10ms to 30ms in our experiments and is essentially the end-to-end latency at a client. When a backup fails (scenario 2), the primary removes the backup from the membership, and there is no negative impact on the end-to-end latency at a client. When a (new or restarted) replica is introduced into a group (scenario 3), it is added as a backup. Even though a state transfer is necessary to bring the new replica up-to-date, the state transfer takes place off the critical path and, thus, the impact on the end-to-end latency at a client is minimal.

VII. Related Work

The LLFT middleware is a software-based approach to fault tolerance like the Delta-4 system [1]. With LLFT, applications programmed using TCP socket APIs, or middleware such as Web Services, can be replicated without modifications and with strong replica consistency. LLFT is designed for distributed applications deployed within a cloud computing environment, rather than over the wide-area Internet. Other research that provides fault tolerance for Web Services exists [4], [5], [6], [7], [8], [9], [10].

The LLFT middleware is transparent to both the application and the operating system, like the TFT [11] and the Hypervisor [12] systems. However, those systems differ from LLFT in the way in which they achieve transparency. 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 middleware can be dynamically inserted (removed) into (from) a running application process using operating system facilities.

The LLFT middleware provides strong replica consistency based on virtual synchrony [2], even if the primary or a backup fails, using the Virtual Determinizer Framework and piggybacking mechanisms. The Delta-4 system [1] uses a separate message, instead of piggybacking, to transmit ordering information from the primary to the backups. Thus, the primary must wait until all of the backups have explicitly acknowledged the ordering notification before it sends a message, which can reduce system performance.

Atomic multicast protocols that deliver messages reliably and in the same total order, such as Isis [2], Amoeba [13], and Totem [14], have been used to maintain strong replica consistency in fault-tolerant distributed systems. However, those protocols introduce delays in either sending or delivering a message. The LLFT Messaging Protocol does not incur such delays, because the primary in a group makes the decisions on the order in which the operations are performed and the ordering information is reflected to the backups in the group.

Most existing membership protocols [2], [14], [15] employ a consensus algorithm to achieve agreement on the membership. Achieving consensus in an asynchronous distributed system is impossible without the use of timeouts to bound the time within which an action must take place. Even with the use of timeouts, achieving consensus can be relatively costly in the number of messages transmitted, and in the delays incurred. To avoid such costs, LLFT uses a novel Leader-Determined Membership Protocol that does not involve the use of a consensus algorithm.

Membership protocols for group communication systems, such as Isis [2] and Totem [14], use the term *view change* to represent a change in the membership of a group. In particular, each successive membership, which involves addition (removal) of a member, constitutes a new view. In LLFT, only membership changes that correspond to a change of the primary constitute a new view, which we refer to as a *primary view change*. In typical group communication

systems, the membership is known more widely than by only the members in the group. In LLFT, only the primary and backups in the group need to know the membership of the group.

The LLFT middleware includes a novel generic Virtual Determinzer Framework to capture, transmit and execute ordering information for non-deterministic operations in the applications. Other researchers [16], [17], [18] have addressed the need to sanitize non-deterministic operations, to achieve strong replica consistency for active replication, rather than for leader/follower replication. The non-deterministic operations handled by LLFT overlap those considered in other systems such as Delta-4 [1], TFT [11] and Hypervisor [12]. LLFT addresses non-determinism caused by multi-threading and socket communication, which those works do not discuss.

VIII. Conclusion

The Low Latency Fault Tolerance (LLFT) middleware provides fault tolerance for distributed applications deployed within a cloud computing or data center environment. Applications programmed using TCP socket APIs, or middleware such as Web Services, can be replicated with strong replica consistency using the LLFT middleware, without any modifications to the applications. Performance measurements show that the LLFT middleware achieves low latency message delivery and membership services for the applications. Its genericity, application transparency, and low overhead make it appropriate for distributed applications deployed within a cloud computing or data center environment.

Acknowledgment

This research was supported in part by NSF grant CNS-0821319.

References

[1] D. Powell, *Delta-4: A Generic Architecture for Dependable Distributed Computing*. Springer-Verlag, 1991.

[2] K. P. Birman and R. van Rennesse, *Reliable Distributed Computing Using the Isis Toolkit*. IEEE Computer Society Press, 1994.

[3] W. R. Dieter and J. J. E. Lumpp, "User-level checkpointing for LinuxThreads programs," in *Proceedings of the 2001 USENIX Technical Conference, June 2001*, Boston, MA, June 2001, pp. 81–92.

[4] K. Birman, R. van Renesse, and W. Vogels, "Adding high availability and autonomic behavior to Web Services," in *Proceedings of the 26th International Conference on Software Engineering*, Edinburgh, Scotland, 2004, pp. 17–26.

[5] K. K. Lau and C. Tran, "Server-side exception handling by composite Web Services," in *Proceedings of the 3rd Workshop on Emerging Web Services Technology*, Dublin, Ireland, November 2008, pp. 30–44.

[6] T. Marian, M. Balakrishnan, K. Birman, and R. van Renesse, "Tempest: Soft state replication in the service tier," in *Proceedings of the IEEE International Conference on Dependable Systems and Networks*, Anchorage, AK, June 2008, pp. 227–236.

[7] L. E. Moser, P. M. Melliar-Smith, and W. Zhao, "Building dependable and secure Web Services," *Journal of Software*, vol. 2, no. 1, pp. 14–26, February 2007.

[8] J. Salas, F. Perez-Sorrosal, M. Patino-Martinez, and R. Jimenez-Peris, "WS-Replication: A framework for highly available Web Services," in *Proceedings of the 15th International Conference on the World Wide Web*, Edinburgh, Scotland, 2006, pp. 357–366.

[9] G. Santos, L. Lung, and C. Montez, "FTWeb: A fault-tolerant infrastructure for Web Services," in *Proceedings of the IEEE International Enterprise Computing Conference*, Enschede, The Netherlands, September 2005, pp. 95–105.

[10] W. Zhao, H. Zhang, and H. Chai, "A lightweight fault tolerance framework for Web Services," *Web Intelligence and Agent Systems: An International Journal*, vol. 7, no. 3, pp. 255–268, 2009.

[11] T. C. Bressoud, "TFT: A software system for application-transparent fault tolerance," in *Proceedings of the 28th IEEE International Conference on Fault-Tolerant Computing*, Munich, Germany, June 1998, pp. 128–137.

[12] T. C. Bressoud and F. B. Schneider, "Hypervisor-based fault tolerance," *ACM Transactions on Computer Systems*, vol. 14, no. 1, pp. 80–107, February 1996.

[13] M. F. Kaashoek and A. S. Tanenbaum, "Group communication in the Amoeba distributed operating system," in *Proceedings of the 11th IEEE International Conference on Distributed Computing Systems*, Arlington, TX, May 1991, pp. 222–230.

[14] L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia, and C. A. Lingley-Papadopoulos, "Totem: A fault-tolerant multicast group communication system," *Communications of the ACM*, vol. 39, no. 4, pp. 54–63, April 1996.

[15] Y. Amir, D. Dolev, S. Kramer, and D. Malkhi, "Membership algorithms for multicast communication groups," in *Proceedings of the 6th International Workshop on Distributed Algorithms, Lecture Notes in Computer Science 647*, Haifa, Israel, November 1992, pp. 292–312.

[16] C. Basile, Z. Kalbarczyk, and R. Iyer, "Active replication of multithreaded applications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 17, no. 5, pp. 448–465, 2006.

[17] R. Jimenez-Peris and S. Arevalo, "Deterministic scheduling for transactional multithreaded replicas," in *Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems*, Nuremberg, Germany, October 2000, pp. 164–173.

[18] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith, "Enforcing determinism for the consistent replication of multithreaded CORBA applications," in *Proceedings of the 18th IEEE Symposium on Reliable Distributed Systems*, Lausanne, Switzerland, October 1999, pp. 263–273.