Tolerating Client and Communication Failures in Distributed Groupware Systems

Hyong Sop Shim and Atul Prakash
Department of Electrical Engineering and Computer Science,
University of Michigan, Ann Arbor, MI 48109-2122 USA
E-mail: {hyongsop,aprakash}@eecs.umich.edu

Abstract

If a groupware system is to be effectively used, especially over a wide-are network such as the Internet, where the quality of networking and computing resources are unpredictable, it should allow clients to tolerate client, link, and server failures. In particular, clients should be able to join groups and transfer groups' current state in the presence of most client and link failures. In order to reduce usage overhead, disconnected clients should also be able to rejoin groups without having to restart from scratch. Furthermore, lock management and group membership should tolerate transient failures in the system. In this paper, we introduce the notion of stateful group communication, which frees clients of administrative management of shared application state and allows fault-tolerant group join, state transfer, and rejoin. Stateful group communication is incorporated in Corona, a general-purpose, group communication service provider. In order to allow groups to tolerate transient failures, Corona also provides locks with grace period and group membership notification services that are based on client connection status. In this paper, we present and discuss Corona's fault-tolerant services.

1. Introduction

The goal of a groupware system is to enable groups of geographically distributed users to collaborate over common tasks over distance and/or time. A synchronous groupware system requires users to be present at their respective sites at the same time, whereas an asynchronous groupware system allows users to work on common tasks at different times. In this paper, we address issues in making synchronous groupware systems fault-tolerant against client, network link, and server failures. In particular, we are only concerned with fail-stop failures [15] of clients and servers. A failed component stops operating until it is restarted, and operational components can detect failures of other components in the system. We also assume that a network link is reliable and FIFO and may only fail by crashing. Two components connected via a failed link cannot exchange messages until a new link is established.

In a groupware system, users are often impatient with any delays due to network congestion, slow links, failed processes, etc. Therefore, if a groupware system is to be effectively used, especially over a wide area network such as the Internet and World Wide Web, where the quality of networking and computing resources are unpredictable, it should allow clients to tolerate client and link failures and server crashes.

In particular, a client should be able to *join* its group in the presence of client and link failures. In addition, existing clients should be able to continue their operations while a new client is joining their group. A group of clients may also share some application state. In such a case, a new client joining the group should be *transferred* the group's current state, even in the presence of client and link failures. Clients may get disconnected due to server crashes and/or link failures in the system. In such a case, a client should be allowed to *rejoin* its groups without having to start over from scratch, which may present substantial usage overhead. A rejoined client should also be able to synchronize with the current group state with minimal overhead. In a groupware system, some clients may provide application-specific services to other clients based on group membership. Therefore, the groupware system should allow such service providers to tolerate transient membership changes due to transient client and link failures. In order to increase application responsiveness, some clients may also require *locks* on shared application objects. In such cases, a groupware system should recover locks from failed lock holders in a timely manner. At the same time, it should tolerate transient failures so that a lock holder's state updates can be salvaged upon rejoin.

Existing group communication subsystems, such as ISIS [4], were originally intended to support building of ro-

bust, replicated *services*. They have been used to manage replicated state among clients in a groupware system [7], but there are limitations of that approach. Although they provide important message ordering and consistent membership view guarantees, the inherent assumption is that members of a replica group are generally available for state transfer, etc. Our experience with groupware systems in wide-area networks is that clients, such as those run from browsers, can often crash, be frequently stopped or terminated by users (such as by closing the browser window), or have poor connectivity to the network. If the client (a replica group member) chosen for state transfer experiences a link failure or crashes, the new client would have to wait until another client is chosen and so on. Thus, frequent membership updates can incur significant performance penalties on clients.

In this paper, we present our approach to providing fault-tolerant services in groupware systems. Specifically, we introduce the notion of *stateful group communication*, in which a group communication subsystem directly manages groups' shared application state and transfers groups' state to new clients. Stateful group communication enables robust group join, state transfer, and rejoin that tolerate client and link failures by delegating shared state management responsibilities to the communication server. In order to support a wide range of groupware applications, the management of shared application state at server is independent of application semantics [12]. In order to protect group state from server crashes, broadcast messages are logged on permanent storage at server. In order to facilitate group rejoin, clients buffer their broadcast messages until acknowledged by the server. Coupled with server message logging, this allows clients to tolerate server crashes and rejoin groups with minimal overhead. This approach has the added benefit of supporting *asynchronous* collaboration where the group state can be transferred from one client to another even when they are not simultaneously connected to the communication subsystem.

Our approach has been designed and implemented in Java as part of the Upper Atmospheric Research Collaboratory (UARC) project [6] and is incorporated into Corona [16] and DistView [13]. Corona is a communication service provider that allows stateful group communication, whereas DistView is a toolkit for building groupware applications on top of Corona services. In addition to group communication services, Corona provides locks with grace period and group membership notification services that reflect *client connection status* to allow groups to tolerate transient failures in the system. Over the past few years, Corona has been successfully used to support team science over a wide area network in UARC.

The rest of the paper is organized as follows. Section 2 presents basic Corona services. Section 3 introduces the notion of stateful group communication and discusses how it allows group join and state transfer to tolerate client and link failures. Section 4 discusses Corona's message logging mechanism. Section 5 discusses locks with grace period. Section 6 discusses Corona's group membership service based on client connection status. Section 7 discusses issues involved in supporting automatic client rejoin and presents its mechanism. Section 8 discusses related work. Section 9 concludes the paper and discusses future work.

2. Corona Group Communication: Basics

Corona supports the notion of a *group*. A group is a set of client applications that are communicating with each other by broadcasting *messages* to the group. A group has various attributes, which include a name and client-defined properties. A group's name uniquely identifies the group in Corona; no two groups can have the same name. A group may also have various properties that are client-defined. A property is a $(name, value)$ pair, $name$ is a the property name, and $value$ is an object that represents the property value. By default, a group does not have any property.

In order to distinguish between different clients, Corona assigns a *client_id* to a client when it joins a group. The *client_id* is also sent to the joining client as part of its `joinGroup()` protocol. The client uses its *client_id* to uniquely identify its own messages.

Corona allows both inter- and intra-group message broadcast. A client broadcasts a message to a group by sending a $bcast(gname, data)$ message to Corona, where $gname$ is the name of the target group, and $data$ is the byte-stream encoding of the message content. Upon receiving a $bcast(gname, data)$ message, Corona broadcasts the message to the members of the target group. Corona allows the message sender to specify whether it wants to receive the message back (sender-inclusive broadcast) or not (sender-exclusive broadcast). Corona also supports $obj\_state\_bcast()$ messages. A $obj\_state\_bcast()$ message contains state update information on a shared object and is described in detail in Section 3.2.

Both $bcast()$ and $obj\_state\_bcast()$ messages have following attributes: *sender_id*, *local_id*, and *global_id*. A broadcast message's *sender_id* uniquely identifies the message sender and is set to the *client_id* of the sender client. The *local_id* is a client-generated sequence number and uniquely identifies a message among other messages broadcast by the same sender. The *global_id* is a Corona-generated sequence number and uniquely identifies the message among all the broadcast messages in the system. The *global_id* also indicates the receive order at Corona. For messages, m_1 and m_2, if $global\_id(m_1) < global\_id(m_2)$, then Corona received m_1 before m_2. No two messages can have the same *global_id*. The sender client sets the

sender_id and *local_id* of a message when it sends the message to Corona, whereas Corona sets the *global_id* of a message when it receives the message.

3. Stateful Group Communication

A group of clients often share some application state for their tasks, e.g., shared documents. In a majority of existing groupware systems, clients assume state transfer responsibilities in that an existing client is chosen to transfer the current shared state to a new client . In this protocol, the new client may experience delay due to the failure of the chosen client or its network link to the system. This is especially true when groups collaborate over a wide-area network such as the Internet, in which the quality and reliability of network connectivity and host machines are unpredictable. Another problem is that the new client may not able to receive application state at all. It is possible that all existing members experience failures while the new member is still joining. Then there would no client left to transfer application state. One possible solution is to store shared application state in a known location on disk. However, the new client may not be able to access the disk, especially when working over a wide-area network.

In order to provide fault-tolerant group join and state transfer services, Corona directly manages a group's shared application state and transfers the group's state to new clients. Because no existing client participates, a new client is protected against client and link failures. Of course, either Corona itself may crash or the network link between Corona and the new client may fail during this protocol. However, the idea is that as a service provider, Corona can be made more reliable than client applications and that its runtime environment can be better controlled, e.g., resources can be dedicated. If a failure does occur, the client can rejoin the group through Corona's automatic client rejoin mechanism as discussed in Section 7. In case of a crash, Corona can still recover the group's state via its message logging mechanism as discussed in Section 4.

3.1. Stateful and Stateless Groups

In Corona, a group that shares application state is called a *stateful group*. Figure 1 graphically illustrates stateful groups. Note that each group member has a copy of shared application state due to Corona's state transfer. This allows a client to update its local copy of the shared state first (assuming the client has appropriate locks (See Section 5)) and then broadcast the updates. This increases application responsiveness, a critical performance factor in an interactive collaboration environment.

Corona also supports *stateless* groups. A stateless group is a communication group, in which group members do not share application state for their work and communicate by broadcasting only $bcast()$ messages.

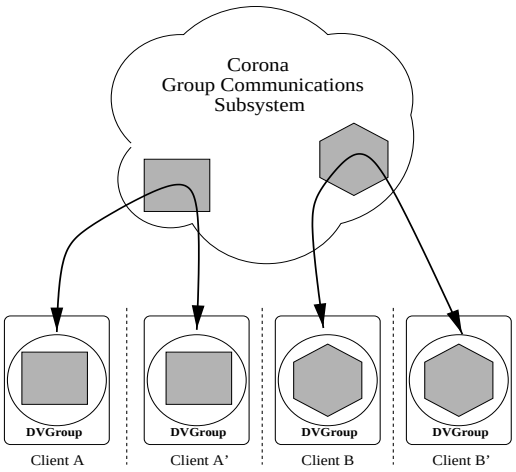

Figure 1. Corona stateful groups. Clients A and A' are in a stateful group and are sharing some application state, identified by a filled rectangle, whereas Clients B and B' are in another stateful group sharing some application state, identified by a filled polygon. DVGroup is a DistView toolkit [13] component that provides interfaces for Corona services, e.g., groupJoin(), groupLeave(), and bcast().

3.2. State Update Messages

In Corona, a stateful group's shared application state consists of a number of application objects, each of which has a client-generated identifier, $shared\_obj\_id$. The $shared\_obj\_id$ of a shared object is known to Corona only when it receives from a client a broadcast message, $obj\_state\_bcast(id, type, data)$ message, where id is the $shared\_object\_id$ of the object, $type$ specifies the type of state update, and $data$ is a byte-stream encoding of state update. $type$ can be either INC or NEW. An $obj\_state\_bcast(id, INC, data)$ message contains an incremental state update in $data$, whereas an $obj\_state\_bcast(id, NEW, data)$ message contains the new state of the specified object in $data$. In order to broadcast an $obj\_state\_bcast()$ message to a group, the sender should be a member of the group.

In addition to $obj\_state\_bcast()$ messages, Corona allows authorized clients to checkpoint stateful groups' state by broadcasting $group\_state\_bcast(data)$ messages, where $data$ contains the latest state of each shared object. That is, $group\_state\_bcast(data)$ messages provide a way of updating all the shared objects in a stateful group.

Note that Corona does not get involved in the interpretation of semantics of shared objects. Corona only knows a shared object by its $shared\_obj\_id$ and a series of $obj\_state\_bcast(id_i, type, data)$ messages, where $id_i = shared\_obj\_id$. Furthermore, Corona does not (or cannot) decode the message contents of either $group\_state\_bcast()$ or $obj\_state\_bcast()$ messages. This client-based semantics of shared objects [12] enables Corona to support a wide variety of client applications.

3.3. Corona State Transfer

In order to facilitate application state transfer, Corona *caches* $bcast()$ and $obj\_state\_bcast()$ messages based on message semantics as follows. An $obj\_state\_bcast(id_i, INC, data)$ message contains incremental state information on the specified object. Thus Corona caches an $obj\_state\_bcast(id_i, INC, data)$ message along with the other $obj\_state\_bcast(id_j, type, data)$ messages in the order they are received, where $id_j = id_i$. On the other hand, an $obj\_state\_bcast(id_i, NEW, data)$ message contains the specified object's complete state. Thus when Corona receives an $obj\_state\_bcast(id_i, NEW, data)$ message, it purges from its message cache any $obj\_state\_bcast(id_j, type, data)$ messages, where $id_j = id_i$, and then saves the message. Likewise, upon receiving a $group\_state\_bcast(data)$ message, Corona purges all previously saved $obj\_state\_bcast()$ messages from its message cache before storing the $group\_state\_bcast()$ message. Corona caches $bcast()$ messages in the order they are received. Messages are cached an in-memory list, called *group_msgs*, which Corona maintains for each group in the system.

A stateful group's state is the last $group\_state\_bcast()$ message followed by a list of $obj\_state\_bcast()$ messages. When a client joins a stateful group, Corona takes a snapshot of the group's state from its *group_msgs* list and sends it to the client. If new messages are broadcast during this process, Corona sends the new messages to the new client after the state is transferred.

No state transfer occurs when joining stateless groups. However, Corona caches $bcast()$ messages for stateless groups in order to support automatic client rejoin as discussed in Section 7.

4. Message Logging at Corona

Corona *logs* broadcast messages in order to protect group state information against its own crashes. For each group, Corona maintains a *group_log_file*. Upon receiving a broadcast message for a group, it first logs the message to the group's *group_log_file* and then broadcasts the message to the group members. This is to provide a guarantee that when a client receives a broadcast message from Corona, the message has already been logged. This guarantee is critical to Corona's automatic client rejoin support (see Section 7). In addition, messages are logged at the time they are cached in *group_msgs* lists. This is to minimize the amount of group state that may be lost due to Corona crashes. Alternatively, for less frequent disk access, a periodic dumping of a *group_msgs* list to the corresponding *group_log_file* could have been done, but that risks losing a period's worth of messages. Performance issues can also be addressed by utilizing a memory-based file cache such as Rio [5]. Corona also stores group attributes in *group_log_files*. At each start-up, Corona reads in *group_log_files* in order to restore groups, their attributes, and the contents of their *group_msgs* lists.

Group_log_files are also used to prevent *group_msgs* lists from growing indefinitely. When the size of a *group_msgs* list reaches a predefined threshold, Corona empties the list. The size of the threshold has an impact on Corona's performance in state transfer. A large threshold means that more messages can be cached in memory and allow for fast state transfer without accessing disk. In contrast, a small threshold means that a *group_msgs* list is more frequently emptied, and thus Corona may have to access the group's log file on disk more often. On the other hand, large thresholds demand correspondingly large amounts of system resources and may limit Corona's scalability in terms of the number of clients it can support.

5. Locks with Grace Period

Some clients may require locks in order to provide low response time when updating shared objects. In order to tolerate transient failures of lock holders, Corona associates a lock with a grace period, during which a disconnected lock holder can rejoin its group and salvage its updates on locked objects. If the grace period expires, Corona can recover locks and grant them to other clients. If Corona crashes while a client has a lock, the client should rejoin its group within the lock's grace period after Corona restarts.

In order to provide flexible lock granularity, Corona allows clients to simultaneously lock multiple objects. Simultaneously locked objects constitute a *lock_set*. In order to create a *lock_set*, a client sends a $lock\_request(\{id_1, ..., id_n\})$ message to Corona, where id_i is the identifier of an object to be locked. If no *lock_set* with overlapping elements exist, Corona creates a requested *lock_set* and broadcasts a $lock\_granted(lock\_set\_id, \{id_1, ..., id_n\})$ message to group members, where $lock\_set\_id$ is the Corona-generated identifier of the *lock_set*. Otherwise, Corona sends a $lock\_denied(\{id_1, ..., id_n\})$ message to the requesting client. In order to release a lock on objects, a

client sends a $lock\_release(lock\_set\_id, \{id_1, ..., id_n\})$ message to Corona. Corona removes the specified object identifiers from the specified *lock_set* and then broadcast a $lock\_released(\{id_1, ..., id_n\})$ message to group members. In order to remove a *lock_set*, a client sends a $lock\_release(lock\_set\_id)$ message to Corona. Corona removes the specified *lock_set* and then broadcast a $lock\_released(\{id_1, ..., id_n\})$ message to group members, where id_i is an element of the removed *lock_set*. Corona caches and logs $lock\_granted$ messages and $lock\_released$ messages to a group's *group_msgs* list and *group_log_file* respectively.

By default, a group does not have a *lock_set*. In order to broadcast a $obj\_state\_bcast()$ message, the sender should have a lock on the specified object, or the object should not be locked.

6. Group Membership

In Corona, some group members may provide application-specific services to other group members and maintain application-dependent data. In such a case, application service providers may make data management decisions based on group membership. Therefore, group membership service should allow application service providers to tolerate transient client and link failures. Otherwise, an application service provider may update its database based on incorrect membership information and can no longer provide proper services. For example, in UARC, Room Manager maintains a database of user locations in Session Manager [8] based on the membership of a predefined group in Corona. Without support for tolerating transient failures, Room Manager would have to update its database every time the group membership changes. This can incur significant usage overhead and performance penalties due to frequent database updates in Room Manager as temporarily disconnected Session Managers rejoin the group.

In order to tolerate transient client/link failures, Corona supports three types of clients in a group: *member*, *disconnected_member*, and *non_member*. When a client joins a group, it becomes a *member*, and Corona broadcasts to other clients in the system a $new\_member\_notif(gname, client\_id)$ message, where $gname$ is the name of the group the client joins, and $client\_id$ is the client's unique identifier assigned by Corona upon join. When a *member* is disconnected due to a client and/or link crash, the client becomes a *disconnected_member*, and Corona broadcasts a $disconnected\_member\_notif(gname, client\_id)$ message. If a *disconnected_member* does not rejoin the group within a predefined timeout period, it becomes a *non_member*, and Corona broadcasts a $non\_member\_notif(gname, client\_id)$ message. A *member* can also become a *non_member* by explicitly leaving the group.

7. Automatic Client Rejoin Support in Corona

Automatic client rejoin support enables a disconnected client to wait until a network link can be established to Corona, rejoin its group(s), synchronize its shared application state with that of the group, if needed, and continue its operations, without having to restart from scratch. Avoiding unnecessary client restarts minimizes usage overhead as users are freed from having to go through a registration process and (possibly manually) recover shared state after a restart. In this section, we discuss our approach to supporting automatic client rejoin in Corona.

There are two issues in supporting automatic client rejoin. First, a rejoining client should be transferred any new messages broadcast while it was disconnected. This allows the client to synchronize its state with the rest of the group. An alternative is to blindly transfer entire broadcast messages to the rejoining client. But this would be wasteful, especially when the group has a large number of messages.

Second, the client may have to re-broadcast some of its own messages upon rejoin. It is possible that the client experienced a link failure as it was broadcasting messages, that Corona crashed before it logged the client's messages, or that the user may continue to work while disconnected. In the last case, the user assumes the risk of losing his or her work due to conflicts. A different approach would be to rebroadcast all of the client's messages upon rejoin. However, this approach requires Corona to detect and reject duplicate messages. This places extra overhead on Corona.

7.1. Basic Approach

During normal operation, a client buffers its own messages in an in-memory list, called *buffered_client_msgs*. A message remains in the buffer until Corona acknowledges the message as follows. If the message is broadcast sender-inclusively, Corona sends the original message back to the client. If the message is broadcast sender-exclusively or broadcast to another group, Corona sends a special ack message to the client. The ack message contains the *sender_id* and *local_id* of the original message.

In addition, a client keeps track of the *global_id* of the last broadcast message from Corona. When rejoining, the client presents this information to Corona, which then can determine a sequence of messages that should be transferred to the client.

7.2. Automatic Client Rejoin Mechanism

A rejoining client presents $last\_global\_id$, the *global_id* of the last broadcast message from Corona. Corona then determines the sequence of messages $M = \{m_1, ..., m_n\}$, where $m_i \in M$ and $global\_id(m_i) > last\_global\_id$, and sends M to the client.

Upon receiving M, the client processes each message, $m_i \in M$ as follows. The goal is to determine whether or not any local updates that the client made while disconnected would conflict with the group's state. A conflict results if the client updated an object, while being disconnected, that has since been locked by another client; note that server can take away a lock from a disconnected client after its grace period expires. If a conflict is found, the client is required to join the group as a new client. This ensures that the client's state is consistent with the rest of the group. An alternative would have been to undo the conflicting local updates. With this approach, it may be possible to salvage local updates that do not conflict with the group's state. However, it requires the client to be prepared for undo even when it has locks. We are currently investigating ways to providing transparent system-level undo support for shared objects.

First, the client checks if m_i is a locally generated message. If so, the client removes the corresponding buffered message. This accounts for the locally generated messages that are received by Corona but not yet acknowledged.

If m_i is a $lock\_granted(lock\_set\_id, S)$ message or a $lock\_released(S)$ message, where $S = \{id_1, ..., id_n\}$, it means that another client has acquired and/or released locks on the specified objects while the client was disconnected. If the client finds in its *buffered_client_msgs* list any $object\_state\_bcast(id_i, type, data)$ message, where $id_i \in S$, then the client empties the *buffered_client_msgs* list, deletes M, and joins the group as a new member.

If m_i is a $object\_state\_bcast(id_i, type, data)$ or a $bcast$ message, the client processes m_i as it would a regular broadcast message. Note that m_i does not conflict with any messages in the client's *buffered_client_msgs* list. If the client had a lock on the specified object prior to getting disconnected, then the presence of m_i means the the client has failed to rejoin the group before the lock's grace period was expired, and Corona granted the lock to some other client. As such, the client would have processed a $lock\_released$ message as described earlier.

If m_i is a $group\_state\_bcast()$, it means that the group's state has been reset after the client got disconnected. Therefore, the client removes all the messages from its *buffered_client_msgs* list, makes note of m_i's *global_msg_id*, and forwards m_i to the client.

After all the messages in M are processed, the client broadcasts any remaining messages in its $buffered\_client\_msgs$. If Corona has received new broadcast messages from other clients after it has sent M to the rejoining client, and these messages conflict with the client's cached messages, Corona would detect conflicts and take appropriate actions as it would with regular broadcast messages.

A failure can occur while a client is rejoining; the rejoining client may experience a link failure and/or Corona may crash. In such a case, the client would try again. If the client receives M from Corona before getting disconnected again, it would process the messages in M and then try to reconnect. If Corona crashes, it would restart, read in *group_log_files*, and then allow rejoins.

8. Related Work

Lotus Notes is an *asynchronous* groupware system that allows clients to be disconnected from the system, work off-line, and reconnect to the system at a later time. Notes does not support *synchronous* collaboration in which participants work together at the same time. On the other hand, Corona is primarily a synchronous groupware system. Although Corona's automatic client rejoin mechanism does allow users to work off-line, it does not have as extensive merge support as in Notes.

In the domain of synchronous groupware systems, we are not aware of any other systems that provide system-level support for automatic client rejoin. In its functionality, Lotus' NSTP [12] most closely resembles Corona. Both systems advocate providing system-level services for managing shared objects and argue for client-based semantics of shared objects. However, NSTP lacks the notion of state transfer (clients acquire remote references to shared objects) and does not support the notion of persistent shared objects. On the other hand, Corona does not support NSTP's Facade-like facilities for browsing shared objects.

Other groupware systems, such as Groupkit [14] and Jupiter [11], also provide an infrastructure for conference management and application sharing. However, they lack system-level support for dynamic shared object specification, group state transfer, and persistent groups, the last of which is made possible by Corona's message logging.

Group communication subsystems such as ISIS [4], Transis [1], and Consul [10] provide an infrastructure for building distributed and reliable services on top of their message broadcasting and membership services. Although they provide message ordering and consistent membership view guarantees, they lack the notion of a stateful group communication of Corona. Instead, they leave the responsibility of application state management to clients. Such a client-based approach may not be suitable for building groupware applications, where clients are free to leave the group at any time. They are better suited to building replicated services [3, 9, 2], where service failures are expected

to be rare. In our experience with groupware systems over the Internet, it is much easier to make servers reliable, compared to clients, simply because resources can be devoted to keep servers running all the time.

9. Conclusion and Future Work

We introduced the notion of stateful group communication as means of providing fault-tolerant services in groupware systems. By freeing clients of the administrative managements of shared application state, stateful group communication allows robust join, state transfer, and rejoin services that tolerate client/link failures and server crashes. Stateful group communication is incorporated in Corona, a general-purpose, group communication service provider. In order to allow groups to tolerate transient failures, Corona provides locks with grace period and group membership notification services that are based on client connection status. Corona has been successfully used over a wide area network as part of the Upper Atmospheric Research Collaboratory (UARC) project.

Several open issues remain. For example, there are cases in which groups of client applications are inter-related and the proper function of one group depends on other groups. In such a case, when Corona crashes, not only do the members of a group lose connection to Corona, but they also lose connection to the members of related groups. When Corona restarts, the members may require that inter-group relationships are properly restored. Depending on operational semantics, it may not be enough to simply recreate groups and re-broadcast lost messages. Some applications may require a specific order in which inter-related groups are created as well as specific members to be present for proper operations. We are currently exploring these and other related issues.

Acknowledgments

This work is supported in part by the National Science Foundation under cooperative agreement IRI-9216848, by the IBM Research Partnership Award, and an equipment Grant from Intel.

References

[1] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: A Communication Sub-System for High Availability. Technical Report TR CS91-13, Computer Science Dept., Hebrew University, April 1992.

[2] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Robust and Efficient Replication using Group Communication. Technical Report TR CS94-20, Institute of Computer Science, The Hebrew University of Jerusalem, Nov. 1994.

[3] K. P. Birman and T. A. Joseph. Low-Cost Management of Replicated Data in Fault-Tolerant Distributed Systems. *ACM Trans. on Computer Systems*, 4(1):54–70, Feb. 1986.

[4] K. P. Birman and T. A. Joseph. Exploiting Virtual Synchrony in Distributed Systems. In *Proc. of 11th ACM Symp. on Operating Systems Principles*, pages 123–138, Austin, TX, Nov. 1987.

[5] P. Chen, W. Ng, S. Chandra, C. Aycock, G. Rajamani, and D. Lowell. The Rio File Cache: Surviving Operating System Crashes. In *Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[6] R. C. et. al. UARC: A prototype upper atmostpheric research collaboratory. *EOS Trans. American Geophys. Union*, 267(74), 1993.

[7] M. Knister and A. Prakash. Issues in the Design of a Toolkit for Supporting Multiple Group Editors. *Computing Systems – The Journal of the Usenix Association*, 6(2):135–166, Spring 1993.

[8] J. Lee, A. Prakash, T. Jaeger, and G. Wu. Supporting Multi-user, Multi-applet Workspaces in CBE. In *Proc. of the Sixth ACM Conference on Computer-Supported Cooperative Work*. ACM Press, Nov. 1996.

[9] S. Mishra, L. L. Peterson, and R. D. Schlichting. Implementing Fault-Tolerant Replicated Objects Using Psync. In *Proc. of IEEE 8th. Symp. on Reliable Distributed Systems*, pages 42–52, Seattle, WA, Oct. 1989.

[10] S. Mishra, L. L. Peterson, and R. D. Schlichting. Consul: A Communication Substrate for Fault-Tolerant Distributed Programs. *Distributed Systems Engineeering Journal*, 1(2):87–103, Dec. 1993.

[11] D. Nichols, P. Curtis, M. Dixon, and J. Lamping. High-Latency, Low-Bandwidth Windowing in the Jupiter Collaboration System. In *Proceedings of UIST '95*, Pittsburgh, PA, 1995.

[12] J. F. Patterson, M. Day, and J. Kucan. Notification Servers for Synchronous Groupware. In *Proc. of the Sixth ACM Conference on Computer-Supported Cooperative Work*. ACM Press, Nov. 1996.

[13] A. Prakash and H. Shim. DistView: Support for Building Efficient Collaborative Applications using Replicated Objects. In *Proc. of the Fifth ACM Conf. on Computer Supported Cooperative Work*, pages 153–164, Chapel-Hill, NC, Oct. 1994.

[14] M. Roseman and S. Greenberg. GroupKit: A groupware toolkit for building real-time conferencing appliations. In *Proceedings of the Fourth Conference on Computer-Supported Cooperative Work*, pages 43–50, Toronto, Canada, October 1992.

[15] F. B. Schneider. Byzantine generals in action: Implementing fail-stop processors. *ACM Trans. on Computer Systems*, 2(2):145–154, 1984.

[16] H. Shim, R. Hall, A. Prakash, and F. Jahanian. Providing Flexible Services for Managing Shared State in Collaborative Systems. In *Proceedings of the ECSCW European Conference on Computer Supported Cooperative Work*, pages 237–252. Klewar Academic Publishers, 1997.