Introducing Fault-Tolerant Group Membership Into The Collaborative Computing Transport Layer

R. J. Loader†, J. S. Pascoe† and V. S. Sunderam‡

†Department of Computer Science
The University of Reading
United Kingdom
RG6 6AY
{Roger.Loader | J.S.Pascoe}@rdg.ac.uk

‡Math & Computer Science
Emory University
Atlanta, Georgia
30302
vss@mathcs.emory.edu

Abstract. In this paper we introduce the novel election based fault tolerance mechanisms recently incorporated into the Collaborative Computing Transport Layer (CCTL). CCTL offers the atomic reliable multicast facilities used in the Collaborative Computing Framework (CCF). Our approach utilizes a reliable IP multicast primitive to implement two electorial algorithms that not only form consensus, but efficiently deliver a compact matrix based view of the network. This matrix can subsequently be analyzed to identify specific network failures (e.g. partitioning). The underlying premise of the approach being that by basing fault tolerance on a reliable multicast primitive, we eliminate the need for specific keep-alive packets such as heartbeats.

1 Introduction

The *Collaborative Computing Frameworks (CCF)* [2] is a suite of software systems, communications protocols, and methodologies that enable collaborative, computer-based cooperative work. CCF constructs a virtual work environment on multiple computer systems connected over the Internet, to form a collaboratory. In this setting, participants interact with each other, simultaneously access or operate computer applications, refer to global data repositories or archives, collectively create and manipulate documents spreadsheets or artifacts, perform computational transformations and conduct a number of other activities via telepresence. CCF is an integrated framework for accomplishing most facets of collaborative work, discussion, or other group activity, as opposed to other systems (audio tools, video/document conferencing, display multiplexers, distributed computing, shared file systems, whiteboards) which address only some subset of the required functions or are oriented towards specific applications or situations. The CCF software systems are outcomes of ongoing experimental research in distributed computing and collaboration methodologies.

CCF consists of multiple coordinated infrastructural elements, each of which provides a component of the virtual collaborative environment. However, several of these subsystems are designed to be capable of independent operation. This is to exploit the benefits of software reuse in other multicast frameworks. An additional benefit is that individual components may be updated or replaced as the system evolves. In particular, CCF is built on a novel communications substrate called the *Collaborative Computing Transport Layer (CCTL)* [5] and it is this that is the focus of this paper.

CCTL is the fabric upon which the entire system is built. A suite of reliable atomic communication protocols, CCTL supports sessions or heavyweight groups and channels (with relaxed virtual synchrony) that are able to exhibit multiple Qualities of Service (QoS) semantics. Unique features include a hierarchical group scheme, use of tunnel-augmented IP multicasting and a multithreaded implementation. Other novel inclusions are fast group primitives, comprehensive delivery options and signals.

The original CCTL did not incorporate failure resilience and experiences with the system demonstrated this to be a critical requisite of any group communication protocol. To address this, the reliable multicast primitive has been modified to give reports of suspected failure i.e. it has been enhanced to act as a failure detector. Thus every multicast message acts as a probe of the sessions liveness. The main advantage of the approach is that it does not require keep-alive packets (e.g. heartbeats). This not only improves bandwidth utilization, but also increases scalability and reduces latency.

An error monitor protocol was introduced to process failure reports and to appropriately signal a second error handling protocol that an election must take place to form consensus. We base consensus on a novel electorial algorithm that compiles a matrix representation of the networks state. From this, we postulate that certain matrix transformations will identify specific types of network failure.

The remainder of this paper is structured as follows. Section 2 introduces CCTL and outlines its architectural design. Section 3 focuses on the CCTL failure model before section 4 describes the lengths to which the architecture was adapted. Sections 5 and 6 describe the error monitor and error handler protocols before section 7 outlines the role of a failure log and how the votes are returned. Section 8 discusses how the result is calculated. Finally, in section 9 we give our conclusions and outline the future directions of the research.

2 An Introduction To CCTL

CCTL is the communication layer of the CCF and as such it provides channel and session abstractions to clients. At its lowest level, CCTL utilizes IP multicast whenever possible. Given the current state of the Internet, not every site is capable of IP multicast over WANs. To this extent, CCTL uses a novel tunneling technique similar to the one adopted in the MBone. At each local subnet containing a group member is a multicast relay. This multicast relay (called *mcaster*) receives a UDP feed from different subnets and multicasts it on its own subnet. A sender first multicasts a message to its own subnet, and then sends the tunneled message to remote mcasters at distant networks. The tunneled UDP messages contains a multicast address that identifies the target subnet. TCP-Reno style flow control schemes and positive acknowledgments are used for data transfer, resulting in high bandwidth as well as low latencies. This scheme has proven to be effective in the provision of fast multiway communications both on local networks and on wide area networks. IP multicast (augmented by CCTL flow control and fast acknowledgment schemes) on a single LAN greatly reduces sender load, thus, throughput at each receiver is maintained near the maximum possible limit (approximately 800 kB/s on Ethernet) with the addition of more receivers. For example, with a 20 member group, CCTL can achieve 84% of the throughput of TCP to one destination. If in this situation TCP is used, the replicated transmissions that are required by the sender cause receiver throughput to deteriorate as the number of hosts increases. A similar effect is observed for WAN's; Table 1 in [6] compares throughput to multiple receivers from one sender using TCP and CCTL.

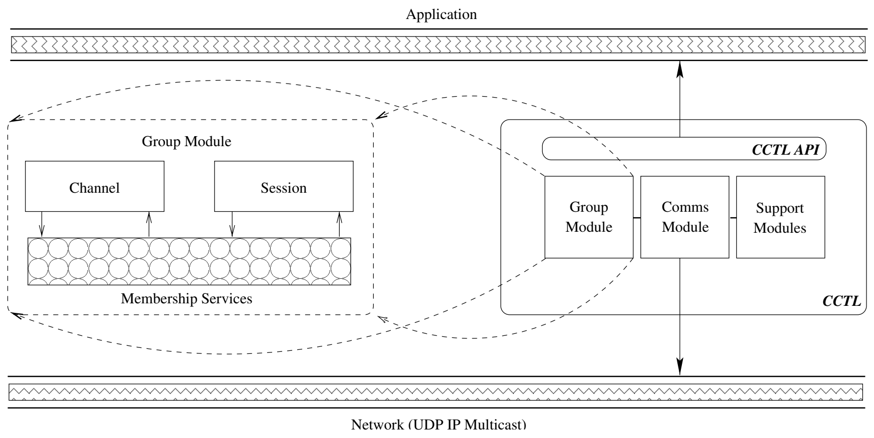

Fig. 1. CCTL Architecture

CCTL offers three types of delivery ordering: *atomic*, *FIFO* and *unordered*. FIFO ordering ensures that messages sent to process q by process p are received in the same order in which they were sent. FIFO guarantees point-to-point ordering but places no constraint on the relative order of messages sent by p and q when received by a third process r.

CCTL offers both reliable and unreliable message delivery. Reliable delivery guarantees that messages sent by a non-faulty process are eventually received (exactly once) by all non faulty processes in the destination set. In a group communication system, this can only be defined in relation to view change operations (membership protocols).

2.1 Architecture

Hierarchical Group Architecture CCTL is logically implemented as a group module, interposed between applications (clients) and the physical network. This module implements the CCTL API and provides session and channel abstractions. Recall that channels are light weight groups supporting a variety of QoS semantics. Note that related channels combine to form a heavy-weight group or session. Recall also that sessions provide an atomic virtually synchronous service called the *default channel*. Sessions and channels support the same fundamental operations (join, leave, send and receive) but many channel operations can be implemented efficiently using the default session channel. Session members may join and leave channels dynamically, but the QoS for a particular channel is fixed at creation. Channels and sessions are destroyed when the last participant leaves.

Fig. 1 shows the CCTL architecture. The group module consists of channel membership, QoS and session sub modules. The channel membership module enforces view change messages (join and leave). The QoS module also provides an interface to lower-level network protocols such as IP multicast or UDP and handles internetwork routing (IP multicast to a LAN, UDP tunneling over WANs).

Several researchers have proposed communication systems supporting light-weight groups. These systems support the dynamic mapping of many light-weight groups to a small set of heavy-weight groups. CCTL statically binds light-weight channels to a single heavyweight session, mirroring the semantics of CSCW environments.

As noted above, CCTL implements channel membership using the default session channel. Session participants initiate a channel view change by multicasting a view change request (join or leave) on the default channel. The channel membership sub module monitors the default channel and maintains a channel table containing name, QoS and membership for all channels in the session. All session participants have consistent channel tables because view change requests are totally ordered by the default channel. This technique simplifies the execution of channel membership operations considerably. For instance, the message ordering imposed by the default channel can be used for ordering view changes. Furthermore, the implementation of channel name services is trivial, requiring a single lookup in the channel table. The architecture of CCTL logically separates channel control transmission (using the default channel) and regular data transmission. This separation increases flexibility by decoupling data service quality from membership semantics.

The hierarchical architecture is also scalable. Glade *et al.* [3] argue that the presence of recovery, name services and failure detectors degrade overall system performance as the number of groups increases. Typically failure detectors periodically poll group members[1]. CCTL performs failure detection on transmission of each multicast message. When a failed process is detected, a unified recovery procedure removes the process from all channels simultaneously, thus restoring the sessions health.

3 Failure And CCTL

Recall that CCTL is implemented as a multithreaded system. However, only the channel sender (CS) thread associated with each channel is of direct interest here; although we acknowledge that the impact of failure during a membership change operation is also an important issue. The sender thread implements a reliable multicast protocol and there is one instance of it for each channel. For each host, the sender thread provides a fault report for every send operation that fails. These operations include the *channel id*, the message sequence number and the *session ids* of those members that have not acknowledged within a time-out [2]. The *session id* of each defaulting session member is encoded as a bit mask.

It is envisaged that members can suffer a number of failures (e.g. process crash, link crash). This can result in either a complete or a *partial failure* of one or more channels. The former problem will be handled by forming a consensus amongst live session members that failed hosts should be removed from the session.The latter problem will be rectified with a second election.

The impossibility result for achieving consensus in asynchronous, message passing systems is well known. Fortunately the addition of even a weak failure detector allows protocols that solve consensus to be developed.

The comprehensive collection and presentation of the scale of the partial failure problem is an important issue. It is possible to provide automatic closing of channels if it is clear that the majority of hosts are reporting a constant stream of irregularities.

[1] E.g. Horus [7, 1] transmits a heartbeat message every two seconds.

[2] The retransmission time-out is actually the slowest round-trip latency for the hosts in the session plus an arbitrary constant

Automatic selective closing of channels will require further investigation as it involves interaction with applications. This is currently the subject of active research.

Symmetric link failures are dealt with using a seniority mechanism. If the session owner can not be reached (i.e. the network has partitioned or the host has suffered a process crash failure), then the most senior of the remaining members can assume the role of the session owner. Should the network subsequently remerge, the most senior of the session owners asserts their authority and any others revert to being standard clients. Typically, a network remerge is detected by the presence of multiple session owners.

The loss of given session member generates errors for any session member that is party to a reliable channel. All live members receive copies of messages but the sender will attempt to contact the failed member. Before the integration of fault tolerance, this resulted in degraded performance that was exasperated as the number of outstanding failures was increased.

4 Adapting The Architecture

It was desirable to investigate if fault tolerance could be introduced without major changes to the existing applications code. The additions to the architecture are given in Fig. 2. Three new threads have been added; these are called the *Error Handler (EH)*, *Error Monitor (EM)* and *Election Timer (ET)*. In addition a new UDP socket, called *FailFd*, is used to allow direct UDP communication between the channel sender, error monitor, error handler and election timer threads. The error detector and error handler also use a new reliable channel called the *fail channel*. The fail channel is similar to the default channel in that every member automatically joins it upon joining the session. Similar to the concept of the session owner one error handler will act as a coordinating master EH(M) and the rest as slaves.

4.1 Monitoring Failure By Augmenting The Sender Thread

The sender thread provides a reliable multicast over CCTL channels. Reliability is ensured by noting acknowledgments from the members of the channel. The details of channel membership are compactly included in the reliable channel message exchanges as an array of unsigned integers. The session id is used to address the bit when its value is changed. When used to represent channel membership, the mask will be termed the

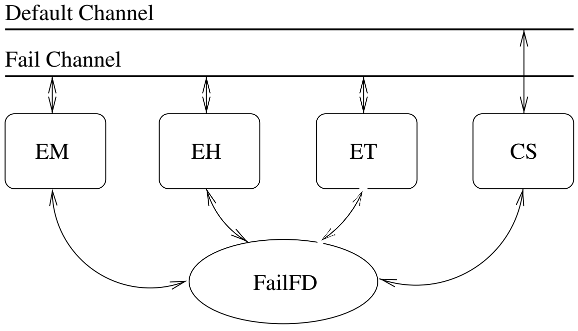

Fig. 2. The Additional Architecture

channel member mask. For the purposes of this discussion membership is indicated by a 1 and non-membership by a 0. When a message is sent, a copy of the channel membership mask is included. This is used by the sender thread to check off the destination session members as they acknowledge. Since the sender's copy is sent via shared memory it is it automatically marked as having acknowledged. In this context the copy has become an *acknowledgment mask*. The state of the acknowledgment mask after the retransmission time-out will be termed the *channel error mask*.

5 The Error Monitor Protocol

The Error Monitor thread (EM), provides first level processing of failure and failure correction reports from the transport mechanism. The main role of the EM is the maintenance of a failure log (that stores failure reports) and a *channel monitor mask* which is used to indicate the failure status of all channels. The log is a shared data structure between the Error Monitor and Error Handler threads. Details of how the log data is recorded are not currently pertinent and are deferred to section 7. It is assumed at this stage that 'add to log' and 'remove from log' functions for a single failure report are present.

Failure reports are transmitted reliably and contain the channel error mask to identify those hosts that did not acknowledge the message. Recall that in CCTL, a dedicated channel (called the fail channel) is provided for the transmission of failure reports and the operation of the election protocols. As each host joins the session, they are automatically admitted to the fail channel. Thus, a message transmitted on the fail channel will be multicast to every host in the session.

On the detection of a failure, the reliable multicast transport mechanism will attempt to resend the message. Each new failure report results in the log being updated. If a failure report is received for a message that has not been logged, then a new entry is made in the log. Otherwise, the appropriate log entry is located, and the corresponding number of failure reports is incremented. Should a host recover and subsequently acknowledge an outstanding message, then the associated number of failure reports is decremented. If the number of reports reaches 0, then the entry is pruned from the log. If the log becomes empty, then an ER_CLEAR message is sent to the Error Handler.

If the number of failure reports exceeds a confirmation threshold[3] for any failure report, the protocol sends an ER_IND message to the Error Handler thread (EH) to signal that a confirmed failure has occurred. Thus, the complete protocol can be described as:

1. A FAIL_REP message results in the log being scanned. If the FAIL_REP corresponds to a known failure, then the number of failure reports associated with that record is incremented. If this value exceeds the confirmation threshold, then this indicates a confirmed failure and an ER_IND message is sent to the EH. Otherwise, a new entry is added to the log and the channel monitor mask is updated to indicate the fault status of the channel.
2. A FAIL_CORR message decrements the number of failure reports associated with a message. If all of the failing hosts subsequently deliver FAIL_CORR messages for a report, then the corresponding entry is pruned from the log. If the log becomes empty then an ER_CLEAR message is sent to the Error Handler.

[3] The value used in our implementation of the approach is 3.

3. The reception of an EL_START message from the EH means that any new FAIL_REP (or FAIL_CORR) messages are suppressed, except for those that result from the EL_PROBE and EL_CALL messages (see below). The occurrence of these events must also generate an ER_IND.
4. An EL_RESULT message contains details of which members are to be removed from the session. In this case, the log is pruned of all entries relating to those failed members.
5. Finally, an EL_END message indicates that the process is complete.

A further role of the EM thread is to govern an election time-out during the voting phase. This topic will be explored in the next section.

6 The Error Handler Protocol

The function of the EH thread is to execute the election based failure recovery protocol. As noted above, there are two versions of the EH protocol; the *master*, referred to as EH(M) and the *slaves* EH(S). Clearly, EH(S) is a subset of EH(M). The master and slaves collectively operate the protocol to request, call and vote in two elections to remove failing hosts and deal with partial failures. Either the EH(M) or any EH(S) can request an election as a consequence of receiving an ER_IND message on the fail channel from its local EM thread[4]. Thereafter the protocol proceeds as follows:

1. The host requesting an election sends a multicast EL_REQ to the EH(M). This also informs the other session members that an election is to take place.
2. The EH(M) responds to an EL_REQ by setting the number of expected failures to zero and multicasting an EL_START followed by an EL_CALL. If there is an ER_IND generated as a consequence of the call then the number of expected failures can be calculated from the channel error mask. A time-out on the voting phase is set.
3. All recipients of the EL_CALL send a multicast EL_PROBE on the fail channel. This can result in fresh failure reports being generated, the purpose of which is for all of the sessions participants to obtain an up-to-date view of the sessions liveness.
4. An EL_PROBE message can generate an ER_IND message from the EM thread because of existing and possibly new failures. Alternatively an EL_PROBE_SUCC message is sent by the EH(M), which indicates that all hosts are capable of responding on the fail channel and that problems previously reported have been resolved.
5. Regardless of whether an ER_IND is received, an EL_RETURN is sent to the error master using a point-to-point message. The return consists of the latest channel error mask plus a serialized form of the session member's failure log.
6. The EH(M) receives the EL_RETURN from each live host. The count is terminated either by all expected returns being received or the EM signals a time-out. The latter is required to guarantee termination when more failures have occurred since the EL_CALL. Note also, that each returned failure log is combined to form the *global failure log*.
7. The EH(M) calculates the result of the election from the data returned (see section 8). The election result is then multicast to the live session members in the form of an EL_RESULT message. Following this, each live member removes those members that are agreed to have failed from its channels.

[4] If at any time, the EH(M) is uncontactable or fails, the role is transferred to the sessions most senior live member and the election is restarted (if one is in progress). Note that the topics of EH(M) failure and its extension to partitioning are covered in more depth in [4].

8. Note that while the election is in progress, further ER_IND messages (other than those expected during the election) are queued by all Error Handlers. On completion of the election, these queues are examined and a session election is called if new failures have occurred.
9. The session election interrogates the global failure log compiled during the first election. Failures on individual channels are considered to ascertain a deeper view of the sessions health. For example, intermittent failures may be evident for a host across the entire session (i.e. for all channels). In this case, the problem can be resolved by transmitting a second EL_RESULT message. Alternatively, a host may be experiencing problems on a subset of the sessions channels. This may be resolved by resynchronizing the affected part of the system.
10. When the algorithm has finished, an EL_END message is multicast to all of the sessions participants informing them that the process is complete.

A more formal definition of the protocols expressed in state event table form is given in [4]. This defines the state variables, the necessary predicates for guarding the atomic actions and gives details of the message formats.

7 The Failure Log And Returning The Vote

The failure log records the stream of failure reports about messages on channels which have not been acknowledged in time. Suppose that the first such report on a channel arrives. Three pieces of information are given: the channel id, the message sequence number and channel error mask with bits set corresponding to the destinations that have not acknowledged the message. If the underlying condition that caused the generation of the message is not cleared, the reliable multicast will send the message again to the tardy hosts until the number of failure reports exceeds the confirmation threshold. Fault reports from the same channel, but with greater message sequence numbers, can also arrive. This is a boon since every failed trial at sending obtains the latest information about the failed hosts in the channel.

It is necessary to keep only the latest copy of the error mask for the channel because it represents the latest information on the session state as far as this channel is concerned. All message sequence numbers are recorded. Suppose that the hosts concerned are only temporarily slow in responding. A stream of FAIL_CORR messages with channel id and sequence numbers will be received. The receipt of a FAIL_CORR on a given channel decrements the total of failure reports associated with a given report. If this count reaches zero then the entry is pruned from the log. If the log is now empty, then the session is considered to be nominal.

8 Producing The Result

The purpose of the election is to assemble from the live members an up to date collective view of the health of the entire session. An initial vote can be taken on the first part of each return, namely the channel error mask resulting from the transmission of the EL_PROBE. This is called the *membership removal election* and it results in any agreed failed members being removed from the session. In the presence of partial failures, this may not repair the session and so the global failure log is then inspected.

Algorithm 1: Membership Removal Algorithm
Code for host EH(M)

```
Initially $V_M^Y \leftarrow V_M^N \leftarrow V_T \leftarrow n_{ret} \leftarrow 0; V_R \leftarrow false; N_s \leftarrow$ number in session

1: while $(n_{ret} < N_s$ - number of estimated failures (from initial ER_IND)$)$
    2: (Receive a time-out; goto 8) $\vee$ (Receive EL_RETURN$_i$ from host $i$)
    3: Convert channel error mask$_i$ to $V_T$ and add failure log$_i$ to global failure log
        4: for (j $\leftarrow$ 0; j $< N_s$; j++)
            5: if $V_T[j] = 1$ then $V_M^Y[j] \leftarrow V_M^Y[j] + 1$
            6: else $V_M^N[j] \leftarrow V_M^N[j] + 1$
    7: nret $\leftarrow$ nret + 1
8: for (j $\leftarrow$ 0; j $< N_s$; j++)
    9: if $V_M^Y[j] \geq \lceil \frac{N_s}{2} \rceil$ then $V_R[j] \leftarrow$ true
    10: else $V_R[j] \leftarrow$ false
11: Send $V_R$ as an EL_RESULT to all hosts
```

Fig. 3. Membership Removal Algorithm

8.1 Membership Removal Election

Once all of the votes have been returned the EH(M) conducts the membership removal election. Declaring a vector V_M^Y for the 'yes' votes, a vector V_M^N for the 'no' votes, a temporary vector V_T, a boolean result vector V_R and an integer n_{ret} to count the number of returns, the informal counting algorithm is given in fig. 3.

The approach taken aims to build a degree of fault tolerance that can allow a minority of simultaneous failures. Suppose there are N_s current session members and n_f failures when the probe took place. The elementary case is defined to be when the group of remaining live members, $N_s - n_f$, is the majority and there are no more failures during the election. In this instance, the value of N_s is reduced by n_f. Now consider that $N_s - n_f$ is still the majority but the vote is split between those voting yes, n_{yes}, those voting no, n_{no}, and those that have failed after the probe but before they can return a result, n_{fail}. Again, the simple case is when the majority of N_s vote yes. A vote where not enough members vote yes, $n_{no} > n_{yes}$, can arise because of a time skew between members and is resolved by holding a session election. The case where not enough votes are cast $n_{yes} + n_{no} < \lceil \frac{N_s}{2} \rceil$, can occur either when an election has taken place in a minority partition or too many extra members have failed during the election. This case is regarded as unrecoverable as is the one where a minority remains.

8.2 Session Election

The execution of the first election may result in the removal of some members. In the presence of partial failures, this may not rectify the problem. An election is now performed using the global failure log to produce an overall picture of the sessions state.

The result of the election is a pair of matrices, where the row index is the channel id, c, and the column index is the host id s. A positive element in the 'yes' matrix signifies that there have been reported errors about the respective member on the corresponding channel. A zero value signifies no reported errors and a negative element indicates that s is not a member of c. Declaring a pair of two dimensional vectors V_S^Y and V_S^N to store the result, we present the session election algorithm in fig. 4.

```
Algorithm 2: Session Election Algorithm
Code for host EH(M)

Initially $V_S^Y \leftarrow V_S^N \leftarrow 0; V_R \leftarrow false; N_s \leftarrow$ number in session

1: for (c $\leftarrow$ 0; c $<$ number of channels; c++)
    2: for (s $\leftarrow$ 0; s $< N_s$; s++)
        3: if (s is a member of c) $\wedge$ (global log shows errors reported about s on channel c)
            then $V_S^Y[s,c] \leftarrow V_S^Y[s,c]+1$
        4: else if (s is a member of c) $\wedge\neg$(global log shows errors reported about s for channel c)
            then $V_S^N[s,c] \leftarrow V_S^N[s,c]+1$
        5: else $V_S^Y[s,c] \leftarrow V_S^N[s,c] \leftarrow -1$
6: for (s $\leftarrow$ 0; s $< N_s$; s++)
    7: if $\forall c \in channel \bullet V_S^Y[s,c] \geq \lceil \frac{N_s}{2} \rceil$ then $V_R[s] \leftarrow$ true
    8: else if $\exists c \in channel \bullet V_S^Y[s,c] \geq \lceil \frac{N_s}{2} \rceil$ then $resynchronize\_channel(c)$
9: if ($V_R$ has been updated) then send $V_R$ as an EL_RESULT to all hosts
```

Fig. 4. Session Election Algorithm

9 Conclusion

This paper describes the novel fault tolerant group membership mechanisms recently incorporated into the Collaborative Computing Transport Layer.

Future work is considering a number of avenues. In the short term we are considering the migration of this approach to wireless environments. It is the opinion of the authors that collaborative computing can benefit greatly from the recent advances in wireless networks and hand-held / wearable computing. When the approach documented here has been fully evaluated, insight will be gained as to how the research should evolve.

References

1. K. P. Birman. *Building Secure and Reliable Network Applications*. Prentice Hall, 1997.
2. S. Chodrow, S. Cheung, P. Hutto, A. Krantz, P. Gray, T. Goddard, I. Rhee, and V. Sunderam. CCF: A Collaborative Computing Frameworks. In *IEEE Internet Computing*, January / February 2000.
3. B. Glade, K. P. Birman, R. Cooper, and R. Renesse. Light weight process groups in the isis system. *Distributed Systems Engineering*, 1:29–36.
4. R. J. Loader, J. S. Pascoe, and V. S. Sunderam. An Electorial Approach to Fault-Tolerance in Multicast Networks. Technical Report RUCS/2000/TR/011/A, The University of Reading, Department of Computer Science, 2000.
5. I. Rhee, S. Cheung, P. Hutto, A. Krantz, and V. Sunderam. Group Communication Support for Distributed Collaboration Systems. In *Proc. Cluster Computing: Networks, Software Tools and Applications*, December 1998.
6. I. Rhee, S. Cheung, P. Hutto, and V. Sunderam. Group Communication Support for Distributed Multimedia And CSCW Systems. In *Proc. 17th International Conference On Distributed Systems*, May 1997.
7. R. van Renesse, K. P. Birman, and S. Maffeis. Horus, a flexible group communication system. In *Communications of the ACM*, April 1996.