Proceedings of the 2003 IEEE
Workshop on Information Assurance
United States Military Academy, West Point, NY, June, 2003

Assuring Consistency and Increasing Reliability in Group Communication Mechanisms in Computational Resiliency

Norka Lucena, Steve J. Chapin, Joohan Lee

Abstract—
The Computational Resiliency library (CRLib) provides distributed systems with the ability to sustain operation and dynamically restore the level of assurance in system function during attacks or failures. In the presence of arbitrary faults, replicated threads need to agree on the values received in order to achieve consistency, when doing group communication in CRLib. To guarantee data integrity and increase reliability, we have implemented a variant of the Lamport-Shostak-Pease oral message algorithm for the Byzantine Generals problem, which provides fuzzy agreement as well as a reduction of the expected communication overhead. Instead of agreeing on the original messages, which could be extremely large, agreement is performed over the 160-bit hashes of normalized messages computed using SHA-1. Performance measurements of applications using CRLib supporting both fail-stop and arbitrary failure models indicate that a reasonable overhead in execution time is worth paying in cases when Byzantine failures are expected.

I. Introduction

Given the increasing number of attacks that result in denial, exploitation, corruption, or destruction of information as well as any activity involving its acquisition, transmission, storage or transformation, it is critical for information systems to have the ability to defend themselves against information warfare (IW) attacks [1]. Researchers define information warfare defense as a continuous process against IW attacks, whose goal is to keep high availability of the system components at any time, and whose phases correspond to a typical protect-detect-react cycle [2]. Therefore, to assure operation, information systems must be resilient, i.e., have the ability to tolerate, recover from, and react to failure and attack.

In general, systems that need to operate under extremely adverse environments use replication of critical data, services and/or resources to increase availability and provide fault tolerance. Although replication of critical information and resources provides graceful degradation of system performance, it is not sufficient to aggressively recover assured operation [3]. The computational resiliency ([3], [4], [5]) model provides distributed information systems with the ability to sustain operation and dynamically restore the level of assurance in system function under IW attacks. This model ensures the restoration of the operational readiness prior to the attacks, subject only to constraints of resource availability.

To realize such a resilient computing model, the authors developed an application-independent, distributed programming middleware library, the Computational Resiliency Library (CRLib), which provides an application-programming interface (API) for concurrent programming, operating in a broad variety of heterogeneous networked architectures. Users of the CRLib only need to specify their particular requirements for reliability. The library transparently takes care of all details regarding communication protocols to achieve on-the-fly replication and reconfiguration. The library also ensures that no communication is lost, that integrity of the process state is maintained, and that, where possible, locality of communication is preserved.

Replication techniques in CRLib are based on the notion of process replication. In particular, due to the concurrent nature of the applications CRLib serves, process replication occurs at the level of thread replication. Several copies of threads are maintained at multiple locations, usually, different computational resources.

To implement replication properly, CRLib organizes threads in groups, which are logical representations of a collection of replicated physical threads. Each thread group, viewed as a single logical unit, hides its internal structure from other groups.

Group communication is based on ordered multicast operations. Although communication details are hidden to the programmer because of the replication transparency provided by the CRLib, multicasting from and to groups of replicated threads leads to replicated messages. A single copy of a thread can receive more than one value, depending on the resiliency (level of replication). In the presence of arbitrary faults, replicated threads need to agree on the values received to achieve consistency.

This study presents a prototype implementation of a so-

This research is sponsored by the Defense Advance Research Projects Agency (DARPA) under contract N66001-99-1-8922 and the Air Force Research Laboratory (AFRL), Information Warfare Directorate, Rome Laboratory, NY.

N. Lucena, S. J. Chapin: Systems Assurance Institute, Syracuse University, 111 College Place 3-114, Syracuse, NY 13244.

J. Lee: School of Electrical Engineering and Computer Science, University of Central Florida, Computer Science Building 227, Orlando, Florida 32816-2362.

ISBN 0-7803-7809-1/03/$17.00 ©2003 IEEE

lution that supports arbitrary failures in CRLib as well as performance measures to quantify the overhead of dealing with such complex failure model while increasing reliability of the applications.

II. Related Work

A. Failure Models

The capability and effectiveness of fault tolerance techniques depend directly on the assumed failure model. Determining how processes and communication links may fail provides understanding of the effect of failures [6] and the suitability of the implemented techniques. Failure models have evolved through the years and become hybrid. They report the presence of distinct fault modes.

Azadmanesh and Kieckhafer [7] summarized the evolution of hybrid fault models and presented a more detailed hybrid model of five modes. In order of complexity, the models they reported are the Single-Mode Byzantine Model, the Two-Mode Meyer and Pradhan Hybrid Model, the Three-Mode Thambidurai and Park Hybrid Model, the Four-Mode Omissive/Transmissive Hybrid Model and the Five-Mode Omissive/Transmissive Model.

The base of all these models is the *Single-Mode Byzantine Model*, which considers only faults of unrestricted behavior, i.e. faulty processes can send conflicting values to different nonfaulty processes. Such behavior is called *Byzantine* or *asymmetric.*

Meyer and Pradhan [8] divided the space of all possible faults into two subspaces: *Benign* faults and *malicious* faults. Benign faults are known to all nonfaulty processes or can be easily determined, such as crash faults. Malicious faults, on the contrary, are not evident and comprise all the other faults.

Thambidurai and Park [9] decomposed the failure space even more. As Meyer and Pradhan [8] did, they partitioned faults into two disjoint subsets: *Non-malicious* faults and *malicious* faults. However, they also partitioned the space of malicious faults into another two disjoint subsets: *Symmetric* faults and *asymmetric* faults. The difference between these last two is based on how the nonfaulty processes perceive their behavior. The behavior of symmetric faults is perceived identically by all nonfaulty processes, i.e. all nonfaulty processes receive exactly the same value. The behavior of asymmetric (Byzantine) faults may be perceived differently by different nonfaulty processes, i.e. the message might not be received identically.

Azadmanesh and Kieckhafer [10] extended the Thambidurai and Park model, which applies only to synchronous systems, introducing two new fault models that encompass faults in asynchronous systems. They consider *symmetric* and *asymmetric* faults, but divide them disjointly into *omissive* and *transmissive* faults. *Omissive symmetric* faults occur when a process fails to deliver any value to any recipient, but the failure is not diagnosed as in benign faults. Under *transmissive symmetric* faults, processes can deliver incorrect values to the receiving processes, but by symmetry, they all receive the same erroneous message. *Omissive asymmetric* faults refer to situations when a process sends a single correct value to some processes and no value to other processes. *Transmissive asymmetric* faults or Byzantine comprise delivering different erroneous messages to different receivers.

Finally, the same authors [7] added *benign* faults to their four-mode model producing the five-mode omissive/transmissive model.

In order to pursue the most general solution to the agreement problem in CRLib, we assume only Byzantine or malicious faults. Therefore, a faulty process may fail in different arbitrary ways: crashing, sending spurious messages to other processes, lying, not responding to received messages correctly, and so on. Moreover, nonfaulty processes do not know which processes are faulty and they have no way to suspect it.

B. Agreement Protocols

B.1 Agreement Problems

An agreement problem is present whenever processes need to agree on a value previously proposed by one or more processes as the correct one [6]. Under the assumption of Byzantine failures, the key idea of the problem is that the nonfaulty processes must be able to reach a common agreement, even when faulty processes are present in the system.

In distributed systems, there are three well-known agreement problems: the Byzantine Agreement problem (also called Byzantine Generals problem), the consensus problem, and the interactive consistency problem [11]. In Byzantine Agreement, only one process provides the value to be agreed on, and all nonfaulty processors have to agree on that value. The process that supplies the initial value is usually distinguished as the commander and the other processes as the lieutenants, according to the Byzantine Generals problem presented by Lamport, Shostak, and Pease [12], [13]. In the consensus problem, every process proposes a single value, its own initial value, and all nonfaulty processes must agree on a single common value. In the interactive consistency problem, every processor also proposes a single value but the goal is to agree on a set of common values, one for each process. Table I presents a summary of the problems based on who initiates the value and the final agreement.

The particular characteristic of these problems is that they are closely related. They can be seen as cases of one another, with the Byzantine agreement problem being the base one. Therefore, solutions for the consensus problems and the interactive consistency problem can be de-

[1] Modified version of Table 8.1: The three agreement problems presented by Singhal and Shivaratri [11].

ISBN 0-7803-7809-1/03/$17.00 ©2003 IEEE

Problem	Byzantine Agreement	Consensus	Interactive Consistency
Who proposes initial value	One processor	All processors	All processors
Final agreement	Single value	Single value	A set of values

TABLE I
Agreement Problems.[1]

rived from solutions to the Byzantine agreement problem, which simplifies implementation and promotes reusability.

B.2 Solutions to the Byzantine Agreement Problem

The Byzantine agreement problem, initially defined and solved by Lamport, Shostak, and Pease [12], [13], has been extensively studied over the years. Every protocol that attempts to solve this problem guarantees that all nonfaulty processors agree in the same value, and that if the processor who initiates the value is nonfaulty, the common value agreed by all processors should be the initial value that was proposed. Moreover, solutions to agreement problems have in common the following system model [11]:

- There are n processors in the system and at most f of them are faulty.
- Processors communicate directly to other processors by message passing. Therefore, a logically fully-connected system is assumed.
- A receiving processor always knows the identity of the sending processor.
- The communication medium is reliable[2], but processes may fail arbitrary.

Another important assumption is a synchronous model of computation, based on rounds. A *round* refers to a computational step where a process receives a message (sent in the previous step), performs the required computations, and sends the resulting messages to other processes (that will be received in the next round). Processes have knowledge about the messages they expect to receive in a round.

Different solutions for the Byzantine agreement problem deal with the trade-off between message complexity and number of rounds in different ways. Garay and Moses [14] listed several solutions presented though the years and compared them in terms of n (total number of processes), required number of rounds for agreement, communication, and computation.

C. Message Digest: SHA-1

A message digest (also known as a hash) is a one-way function that takes an input message and produces an output. The one-way property is what makes a message digest function cryptographically secure. It should be computational infeasible to determine a message, given its message digest, and, similarly, it should be impossible to find two messages that produce the same message digest [15].

There are several algorithms to compute message digests: MD2, MD4, MD5, and SHA, among the well-known ones. In particular, SHA (Secure Hash Algorithm) is a specification for computing a condensed representation of a message or a data file proposed by the National Institute of Standards and Technology (NIST) as part of the Secure Hash Standard (SHS) [16]. There are several versions of the Secure Hash Algorithm: SHA-1, SHA-256, SHA-384, and SHA-512 [17]. They mainly differ in the length of the message digest, ranging from 160 to 512 bits, depending on the algorithm. Nevertheless, currently, the only FIPS-approved [3] algorithm for generating message digests is SHA-1. In addition to that, SHA-1 is the algorithm specified in the Digital Signature Standard (FIPS PUB 186-2) to generate the condensed version of the message to be signed [18] and is the U.S. government's standard hash function.

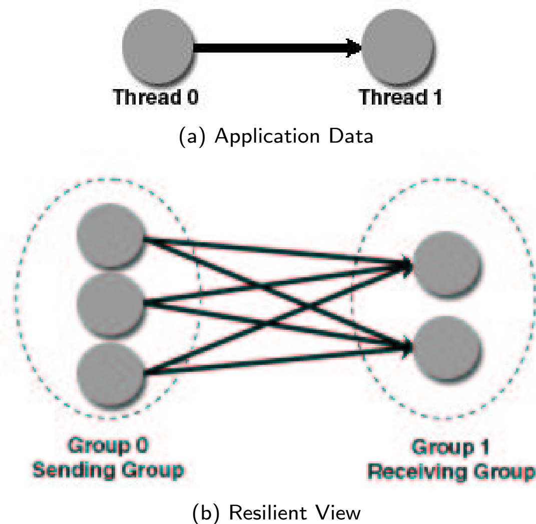
III. Prototype Implementation

A. Need for Agreement in CRLib

In order to provide an application with the ability to sustain operation and dynamically restore the level of assurance in system function during attack, once a programmer has set up the communication structure of the application and the level of resiliency (i.e., number of replicated threads), CRLib supplies means for dynamic reconfiguration and replication of threads. Dynamic reconfiguration of groups of replicated threads involves solving critical issues such as describing and managing the group, detecting a compromise, and ensuring valid program state and communication structure [3]. CRLib addresses those problems through the implementation of three protocols: the group membership protocol, the liveness checking and recovery protocol, and the flow control protocol. The membership protocol provides mechanisms to create threads and to add (join operation) or withdraw (leave operation) threads from a group. The liveness checking and recovery protocol implements an interface to application specific routines for detecting a compromise and the recovery mechanism. The flow control protocol provides communication abstraction based on ordered delivery multicast operations. Details of communications are hidden to the programmer, as it is shown in the Figure 1 where replicated threads multicast messages to threads in the receiving group, but to the application, only one high level message is sent.

[2] The same authors report that, recently, agreement problems have been studied under failures of the communication links only and under both communication and process failures.

[3] NIST cryptographic standards are specified in Federal Information Processing Standards (FIPS) Publications. The term FIPS-approved indicates something (e.g., a cryptographic algorithm) that is either a) specified in a FIPS or b) adopted in a FIPS and specified either in an appendix to the FIPS or in a document referenced by the FIPS.



(a) Application Data

(b) Resilient View

Fig. 1. Group Channel Implementation.

In particular, the flow control protocol ensures reliable delivery of messages in the sense that each receiving thread will get one or more copies of the same message. Figure 2 illustrates this situation. Threads in the sending group multicast their messages to threads in the receiving group. Consequently, any thread there could received the same message several times, as many times as the level of resiliency of the sending group, resiliency level 3 in the example. The receiving thread discards duplicated messages, reorders them, and then sends them to the application level thread.

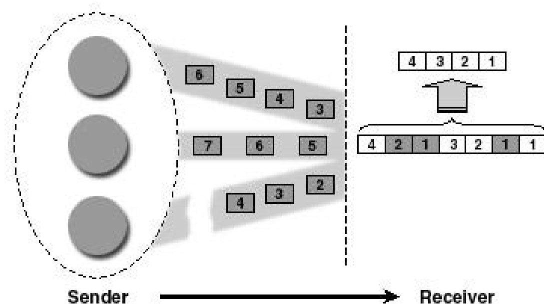


Fig. 2. Flow Control Mechanism.

However, we do not assume an underlying reliable multicast mechanism. Our multicast is implemented within the user process, and at that level is implemented over unreliable unicast mechanisms, making it susceptible to compromise. Because when using CRLib the multicast mechanism is under the control of the sender, it is possible for the sender to achieve an asymmetric or Byzantine fault by dividing the multicast groups at the unicast level.

In addition to that, because replicated threads can be located in different systems, intermediate computations can differ. This, for certain scientific requiring precise numerical values, could produce erroneous results or cause divergence or disagreement on a final value. Figure 3 shows such situation.

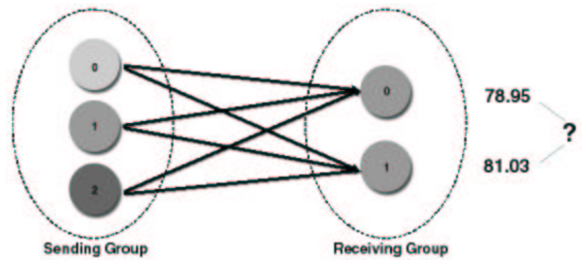


Fig. 3. Possible Scenario where Replicas of Same Thread Receive Different Messages, Leading to Inconsistent Values.

To achieve consistency in the presence of faults (considering each replicated thread in the sending group as a potentially arbitrary faulty thread), receiving threads should exchange their values with other peer threads and relay the values received from them several times. This process of isolating the effects of faulty threads is exactly that of reaching a consensus.

B. Group Communication Cases in CRLib

Based on the resiliency (replication level) of both the sending and the receiving group, there are four different cases of group communication in CRLib. Some of these cases correspond to the well-known agreement problems described in Table I, which allow presenting a solution for the communication problems based on agreement protocols.

B.1 Case 1: Both Resiliency of the Sending Group and Resiliency of the Receiving Group are equal to 1

This is the simplest case. The single copy of the thread in the sending group sends messages to the single copy of the thread in the receiving group (see Figure 4). Since communication is one to one, its solution is trivial: there is nothing to agree on. The single message received is considered valid.

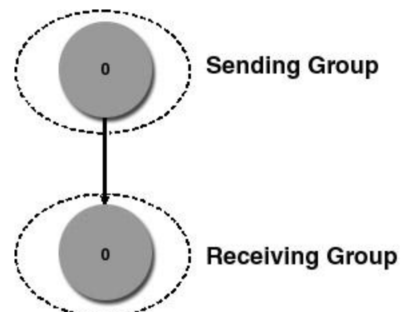


Fig. 4. Case 1: Sending group resiliency == 1 and Receiving group resiliency == 1.

ISBN 0-7803-7809-1/03/$17.00 ©2003 IEEE

B.2 Case 2: Resiliency of the Sending Group is equal to 1, but Resiliency of the Receiving Group is greater then 1

The single copy of the thread in the sending group broadcasts messages to each replicated thread in the receiving group (see Figure 5). Because there is no guarantee that the sending thread is not faulty (and it can therefore be sending spurious messages), threads at the receiving end need to *agree* in the value received. This case corresponds to the Byzantine Agreement problem.

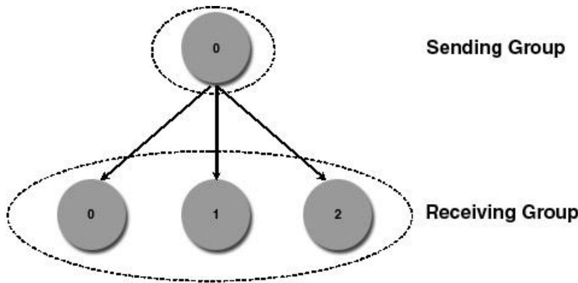


Fig. 5. Case 2: Sending group resiliency == 1 and Receiving group resiliency > 1.

B.3 Case 3: Resiliency of the Sending Group is greater than 1, and Resiliency of the Receiving Group is equal to 1

In this case, the single copy of the thread in the receiving group receives as many messages as the number of replicated threads (level of resiliency) in the sending group, as shown in Figure 6. Although no agreement is needed, the receiving thread still needs to decide which of the message values is valid. For that, a simple voting algorithm satisfies the need.

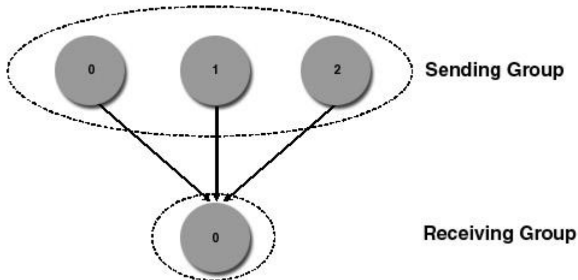


Fig. 6. Case 3: Sending group resiliency > 1 and Receiving group resiliency == 1.

B.4 Case 4: Resiliency of both, Sending Group and Receiving Group is greater than 1

This is the most complex case. Each replicated thread in the sending group broadcasts a message to each thread in the receiving group, as it is illustrated in Figure 7. Thus, threads in the receiving end need to reach agreement on each message received. Therefore, several rounds of agreement are needed— as many as the resiliency level of the sending group (which is three in the example). In addition to that, once replicas in the receiving group have agreed on the values received from each thread of the sending group, they need to decide what message is valid. For that, if each of them executes a voting algorithm they can easily discard any spurious message sent by a faulty thread.

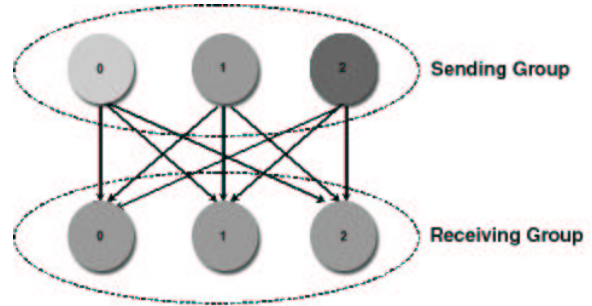


Fig. 7. Case 4: Sending group resiliency > 1 and Receiving group resiliency > 1.

C. Solution Implementation Details

The solution presented in this paper aims to solve the need for reaching agreement and consensus in some of the communication cases of CRLib efficiently and without causing too much overhead. A similar algorithm to the one presented by Lamport, Shostak, and Pease [12] was used to reach agreement, but instead of using the original application messages the described solution uses hashes of those messages. This algorithm is conservative in comparison with more recent algorithms, but makes not assumptions about the type of malicious faults, which is exactly our assumption.

We compute message digests using SHA-1 to reduce the communication overheard when executing the agreement protocol, which is particularly important considering that CRLib can be used for distributed applications that deal which extremely large messages. However, this particular feature is susceptible to the problem of inconsistent values when dealing with heterogeneous machines, and even in some homogeneous cases: floating-point computations can result in slightly differing values. Because of the properties of the hashing algorithm, a one-bit difference between two inputs will produce differing hashes, and there is no correspondence between the difference distance between the inputs and the difference distance between the hashes. Then, reaching agreement or consensus would be almost impossible under such circumstances. To solve this problem, we developed the notation of fuzzy agreement. The fuzziness is eliminated right before computing the message digest. A user-defined application-dependent function normalizes the message to a value within a valid range determined by the user and then, the message is hashed: A simple

example for floating point is to round the input to a precision that is consistently represented across machines. In that way, the probability of obtaining identical hashes for nearly identical messages is much higher.

C.1 Agreement

The implemented algorithm corresponds to the solution with oral messages presented by Lamport, Shostak, and Pease [12]. Assuming that n is the total number of replicas and f the number of faulty replicas, there must be at least $3f + 1$ replicas to cope with f. The characteristics of oral messages are given by the following assumptions:

- Every message that is sent is delivered correctly.
- The receiver replica knows who sent the message.
- The absence of a message can be detected.

Furthermore, the algorithm assumes a function *majority* that takes a set of values $v_1, \ldots, v_{n-1}$ and determines a v, which would usually be the v_i repeated the greatest number of times, but actually it could be the median or any other value depending on the application. The *majority* function is a user-defined function in the implemented solution.

Figure 8 shows in detail the Lamport-Shostak-Pease Oral Messages Algorithm for replicated threads, where the initiator thread is the replica initiating the communication.

Algorithm OM(0)
(1) The initiator thread sends its value to every replicated thread.
(2) Each replica uses the value it receives from the initiator thread, or uses a default value if it receives no value.

Algorithm OM(f), $f > 0$
(1) The initiator thread sends its value to every replicated thread.
(2) For each i, let v_i be the value replica i receives from the initiator thread, or else be a default value if it receives no value. Replica i acts as the initiator thread in Algorithm OM(f - 1) to send the value v_i to each of the n - 2 other replicas.
(3) For each i, and each $j \neq i$, let v_j be the value replica i received from replica j in step (2) (using Algorithm OM(f - 1)), or else a default value if it received no such value. Replica i uses the value $majority\ (v_1, \ldots, v_{n-1})$.

Fig. 8. The Lamport-Shostak-Pease Oral Messages Algorithm.

C.2 Extending Implementation of Group Semantics

Groups of replicated thread in CRLib are open. That is, other threads outside the group may send messages to the group. Initially, CRLib provided only communication mechanisms that allowed replicas from one group to send messages to replicas in a different group but not among peer replicas of the same group. To reach agreement, member threads of the same group need to relay messages among each other. Therefore, additional internal group communication mechanisms were implemented allowing member threads, within the same group, to multicast to themselves.

C.3 SHA-1

The implementation of the SHA-1 algorithm, in accordance with the Secure Hash Standard (FIPS PUB 180-1), is an independent module added to the CRLib for two purposes: one is to produce a unique constant-size version of the message (160 bits) for use in agreement, and the other to facilitate implementation of alternative solutions that require signed messages.

C.4 Fuzziness

The user can provide an application-dependent function that CRLib will use to round the message to a certain threshold defined by the user. This furnishes a means of eliminating the fuzziness that could lead to the impossibility of reaching agreement in a value other than the default one, because of the fact that the resulting message digests are completely different. The performance of this function will depend on the implementation and on the size of the messages.

IV. Experimentation

Singhal and Shivaratri [11] report *time*, *message traffic*, and *storage overhead* as the most commonly used metrics for measuring performance of agreement protocols. Time refers to the time taken to reach an agreement under a particular protocol, that is, the number of rounds needed to reach agreement. Message traffic is usually measured in two ways: as the number of messages exchanged or as the total number of bits exchanged to reach agreement. Storage overhead measures the number of bytes of data stored at the processors when executing the agreement protocol.

The implementation of our solution to agreement problems reduces the number of bits exchanged and stored when doing agreement through a constant 160-bit message size, a significant improvement considering that many distributed applications deal with large messages. The time required to reach agreement of the solution is the same time of the Lamport-Shostak-Pease solution, $f + 1$ rounds as well as the number of messages exchanged, $O(n^f)$. However, our solution also minimizes communication and storage overhead, because the number of bits exchanged and stored is much less in comparison to traditional solutions.

A. Basic Test Application

The basic test application is a very simple program that transfers arrays of characters back and forth. There are no computations performed over those string values. The purpose of such a minimalistic application design was to simplify the measurement of communication overhead. These results closely track those achieved with more complex applications, so we present the minimalistic application for the sake of simplicity.

Results of tests with the number of replicas, n, equal to 4, 7, and 10 (for which a number of faulty threads, f, of 1, 2, and 3, respectively, were assumed) indicate different degrees of overhead depending on the communication case.

Figure 9 compares the performance of the application with different message sizes when a single thread multicasts its value to several replicated threads (see communication

case 2, illustrated in Figure 5). For this particular case, the overhead corresponds to the communication overhead when exchanging hashes in order to reach agreement plus the computation overhead of hashing the message itself. Observed results show that the performance degradation is never greater than 48% when compared to the same application run without any agreement.

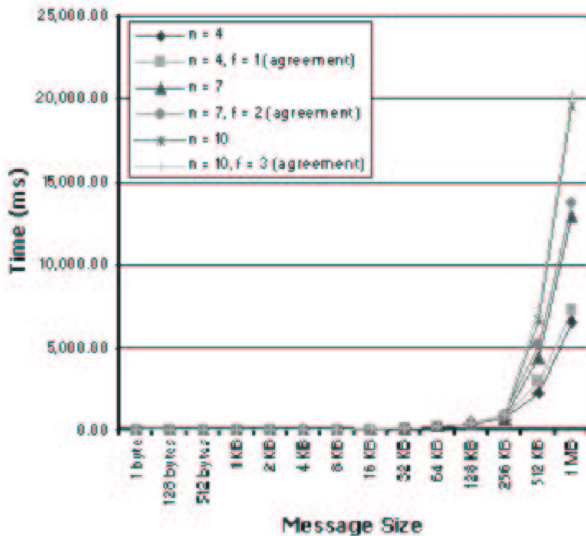


Fig. 9. Execution Time of Basic Application when Reaching Byzantine Agreement (one sending replica, many receiving replicas) with n Total Replicas of which f are Faulty.

Similarly, Figure 10 shows the execution times of the application, with the same message sizes, but when many replicated threads multicast their values to a single thread (see Figure 6). Since no agreement is needed, just a simple voting to select one of the messages, the overhead in this case is minimal and not a product of the communication but of the hashing, rounding, and voting.

On the other hand, Figure 11 illustrates the performance with the same number of replicas and the same message sizes in the most complicated case of communication: when several threads of one group multicast to several other threads in some other group. This is the consensus case shown in Figure 7. Overhead in this case is a sum of the execution time of the hashing function, the communication time of reaching consensus, and the final voting to select which of the agreed messages is the valid one. In particular, this communication overhead is much higher than in other cases because several rounds of agreement need to be performed.

In our approach, there is a tradeoff of speed vs. accuracy when compared to sending the data for agreement rather than the hash in the case of consensus agreement. Because the number of rounds and messages sent is invariant with respect to sending hashes or full messages for agreeemnt, the overhead of sending full messages is roughly equal to the ratio between the hash size and the size of the original

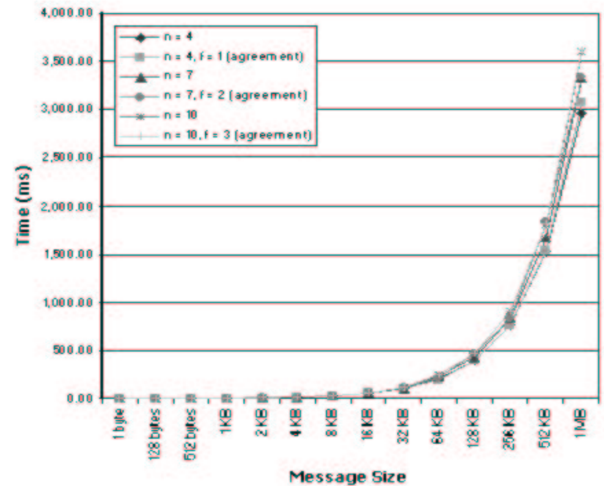


Fig. 10. Execution Time of Basic Application when Voting (many sending replicas, one receiving replicas) with n Total Replicas of which f are Faulty.

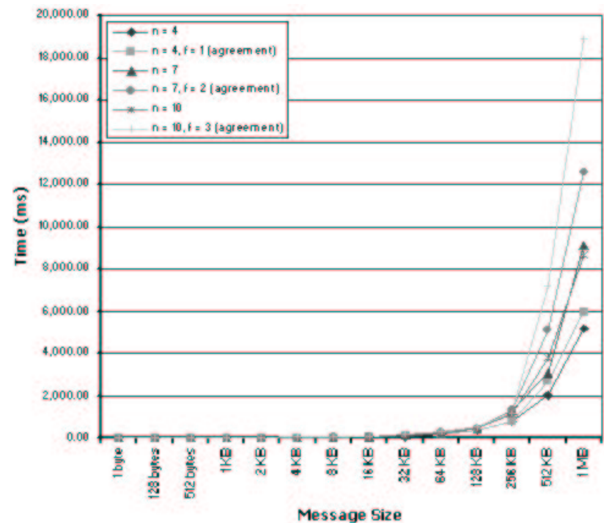


Fig. 11. Execution Time of Basic Application when Reaching Consensus (many sending replicas, many receiving replicas) with n Total Replicas of which f are Faulty.

message. In the case of the SHA-1 algorithm, this overhead for a message M is just $\frac{sizeof(M)-20}{20}$, which scales linearly with the size of the message. For 128-byte messages, the communication load is 5 times higher than with our method; for 4-kilobyte messages, it is more than 200 times higher, and for 1-megabyte messages, more than 52,000 times higher.

Consider that for a replication level of 10, allowing for three failures, the hashed messages took 34 milliseconds for 4K messages, 306 milliseconds for 64K messages, and 19 seconds for 1M messages. If we sent the complete messages, this agreement would take in the neighborhood of 6 seconds for 4K messages, one minute for 64K messages,

and more than an hour for 1-megabyte messages. Considering that scientific calculations typically exchange data on each iteration step and run for thousands to millions of iterations, this overhead can quickly come to dominate the computation time, making it infeasible to run simulations that send large messages.

The cost of this efficiency is the risk that we may refuse to agree in a situation where an alternative method might find agreement. In a contrived example, we have four processes agreeing on a two-element vector, and the rounding function rounds to the nearest integer before computing the hash. With the four vectors as follows:

$v_1 = [0.49 \quad 1.51] \quad v_2 = [0.50 \quad 1.50]$

$v_3 = [0.51 \quad 1.52] \quad v_4 = [0.52 \quad 1.49]$

which, after rounding, become:

$v_1 = [0 \quad 2] \quad v_2 = [1 \quad 2]$

$v_3 = [1 \quad 2] \quad v_4 = [1 \quad 1]$

we will be unable to agree after hashing, because no three hashes will match. However, if we instead agreed on the individual components of the vectors, we would see that three out of the four vectors match on each of the elements, and could agree upon [1 2].

In actual use with three appliction programs, we have not found any problem with accuracy in agreement. We believe the ability to attempt agreement on much larger messages than would otherwise be practicable outweighs the possibility of disagreement on a small subset of the possible messages.

V. Conclusions and Future Work

A solution that provides fuzzy agreement, guaranteeing consistency of the data in the presence of arbitrary faults, when doing group communication in CRLib was implemented and shown to work. Such an implementation is a significant improvement to the library since it makes it much more reliable and ensures data integrity. An innovative approach of agreeing on hash values of the messages, computed using SHA-1, reduces the total number of bits exchanged to reach an agreement. Therefore, the expected communication overhead was minimized. Furthermore, means to eliminate the fuzziness present when doing computations in heterogeneous environments are provided, so digests of similar messages are the same and an agreement could be reached.

A basic test application was developed to measure the overhead in performance when dealing with different communication cases and a variety of message sizes. Results of these experiments indicate that due to the number of rounds of agreement needed to reach consensus, communication is the dominant form of overhead, and our approach is up to 50,000 times more efficient for large messages because of the small constant size of the SHA-1 hash. In addition to that, tests over a real-world application using the CRLib when supporting both fail-stop and arbitrary failure models suggested that the performance overhead was acceptable in cases when any kind of failures can occur.

Our future work will include the use of signed messages in the agreement protocol, thereby constraining on n to be greater than $2m$, rather than greater than $3m$ with unsigned messages.

References

[1] M. C. Libicki, "What is information warfare?." ACIS Paper 3, National Defense University, Fort McNair, D.C., August, 1995. Retrieved on May 05, 2002 from the World Wide Web: http://www.ndu.edu/inss/actpubs/act003/a003.html.

[2] S. Jajodia, C. D. McCollum, and P. Ammann, "Trusted recovery," *Communications of the ACM*, vol. 42, pp. 71–75, July, 1999.

[3] J. Lee, S. Chapin, and S. Taylor, "Computational resiliency," *Journal of Quality and Reliability Engineering International*, vol. 18, pp. 1–15, March 2002.

[4] J. Lee, *Computational Resiliency: Reliable Heterogeneous Applications*. PhD thesis, Syracuse University, 2001.

[5] J. Lee and S. Taylor, "Advances in computational resiliency," in *2001 IEEE Aerospace Conference Proceedings* (2001, ed.), vol. 6, (Big Sky, MT).

[6] G. Coulouris, J. Dollimore, and T. Kindberg, *Distributed Systems: Concepts and design*. Harlow, England: Addison-Wesley, 3rd ed., 2001.

[7] M. H. Azadmanesh and R. M. Kieckhafer, "Exploiting omissive faults in synchronous approximate agreement," *IEEE Transactions on Computers*, vol. 49, pp. 1031–1042, October, 2000.

[8] F. J. Meyer and D. K. Pradhan, "Consensus with dual failure modes," in *Proceeding of the 17th Fault-tolerant Computing Symposium*, pp. 48–54, July, 1987.

[9] P. M. Thambidurai and Y.-K. Park, "Interactive consistency with multiple failures modes," in *Proceedings of the Seventh Reliable Distributed Systems Symposium*, October, 1988.

[10] M. H. Azadmanesh and R. M. Kieckhafer, "New hybrid fault models for asynchronous approximate agreement," *IEEE Transactions in Computers*, vol. 45, pp. 439–449, April, 1996.

[11] M. Singhal and N. G. Shivaratri, *Advanced Concepts in Operating Systems: Distributed, database, and multiprocessor operating systems*. McGraw-Hill, 1994.

[12] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," *ACM Transactions on Programming Languages and Systems*, vol. 4, pp. 382–401, July, 1982.

[13] M. Pease, R. Shostak, and L. Lamport, "Reaching agreement in the presence of faults," *Journal of the ACM (JACM)*, vol. 27, pp. 228–234, April, 1980.

[14] J. Garay and Y. Moses, "Fully polynomial byzantine agreement in t + 1 rounds," in *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing*, (San Diego, CA), pp. 31–41, 1993.

[15] C. Kaufman, R. Perlman, and M. Speciner, *Network Security: Private communication in a public world*. PTR Prentice Hall, 1995.

[16] "Secure hash standard." Federal Information Processing Standard Publication (FIPS PUB 180-1), U.S. Department of Commerce, Technology Administration, National Institute of Standards and Technology, April 17, 1995. Retrieved on January 31, 2001 from the World Wide Web: http://csrc.nist.gov/publications/fips/fips180-1/fips180-1.pdf.

[17] "Secure hash standard." Federal Information Processing Standard Publication (FIPS PUB 180-2), U.S. Department of Commerce, Technology Administration, National Institute of Standards and Technology, 2001. Retrieved on June 01, 2001 from the World Wide Web: http://csrc.nist.gov/encryption/shs/dfips-180-2.pdf.

[18] "Digital signature standard (dss)." Federal Information Processing Standard Publication (FIPS PUB 186-2), U.S. Department of Commerce, Technology Administration, National Institute of Standards and Technology, January 27, 2000.