

An Efficient Fault Tolerance Protocols for Mobile Computing Systems

* Kumar Surender **,Kumar Parveen *** Chauhan R.K

* Lecturer Deptt. Of I.T., H.C.T.M. Kaithal

** Professor Deptt. Of CSE, APIIT, Panipat.

***Chairman Deptt. Comp. Sc. & Application KUK

Abstract:

Fault tolerance is the ability of a system to perform its function correctly even in the presence of internal faults. The purpose of fault tolerance is to increase the dependability of a system. A complimentary but separate approach to increasing dependability is fault prevention. This consists of techniques, such as inspection, whose intent is to eliminate the circumstances by which faults arise. Faults can be classified as *transient* or *permanent*. A transient fault will eventually disappear without any apparent intervention, whereas a permanent one will remain unless it is removed by some external agency. While it may seem that permanent faults are more severe, from an engineering perspective they are much easier to diagnose and handle. The intermittent transient faults that recur often unpredictably are the most problematic. Nowadays, mobile computing is a new software paradigm that is of prime interest in the Information Technology research community. This paper looks at various fault tolerance protocols and issues applicable to mobile devices such as Personal Digital Assistants (PDAs).

Introduction:

A Mobile computing system is a distributed system where some of nodes are mobile computers[1]Mobile computing technology allows transmission of data via a computer system without being connected to a fixed physical link. The nodes have no common clock and no shared memory among them. They communicate each other through messages. Each node operates independently of the others, with occasional asynchronous message communication. The location of mobile computers in the network may change with time. Mobile data communication is a very important and rapidly evolving technology because it allows us to transmit data from remote locations to other fixed or remote locations. It solves the problem of business people on the move. Mobile computing is concerned with the

connection of mobile or portable computers, and with the migration of software from a home platform to a remote one. We need access to computing and communications not only from our home base, but also while we are in transit and when we reach our destination.

Fault-tolerant computing is the art and science of building computing systems that continue to operate satisfactorily in the presence of faults. A fault-tolerant system may be able to tolerate one or more fault-types including -- i) transient, intermittent or permanent hardware faults, ii) software and hardware design errors, iii) operator errors, or iv) externally induced upsets or physical damage. We have different fault tolerant approaches :

Hardware Fault-Tolerance -- The majority of fault-tolerant designs have

been directed toward building computers that automatically recover from random faults occurring in hardware components. The techniques employed to do this generally involve partitioning a computing system into modules that act as fault-containment regions. Each module is backed up with protective redundancy so that, if the module fails, others can assume its function. Special mechanisms are added to detect errors and implement recovery. Two general approaches to hardware fault recovery have been used: 1) fault masking, and 2) dynamic recovery.

Fault masking is a structural redundancy technique that completely masks faults within a set of redundant modules. A number of identical modules execute the same functions, and their outputs are voted to remove errors created by a faulty module. Triple modular redundancy (TMR) is a commonly used form of fault masking in which the circuitry is triplicated and voted. The voting circuitry can also be triplicated so that individual voter failures can also be corrected by the voting process. A TMR system fails whenever two modules in a redundant triplet create errors so that the vote is no longer valid. Hybrid redundancy is an extension of TMR in which the triplicated modules are backed up with additional spares, which are used to replace faulty modules -- allowing more faults to be tolerated. Voted systems require more than three times as much hardware as nonredundant systems, but they have the advantage that computations can continue without interruption when a fault occurs, allowing existing operating systems to be used.

Dynamic recovery is required when only one copy of a computation is running at a time (or in some cases two unchecked copies), and it involves automated self-repair. As in fault masking, the computing system is partitioned into modules backed up by spares as protective redundancy. In the case of dynamic recovery however, special mechanisms are required to detect faults in the modules, switch out a faulty module, switch in a spare, and instigate those software actions (rollback, initialization, retry, restart) necessary to restore and continue the computation. In single computers special hardware is required along with software to do this, while in multicomputers the function is often managed by the other processors.

Dynamic recovery is generally more hardware-efficient than voted systems, and it is therefore the approach of choice in resource-constrained (e.g., low-power) systems, and especially in high performance scalable systems in which the amount of hardware resources devoted to active computing must be maximized. Its disadvantage is that computational delays occur during fault recovery, fault coverage is often lower, and specialized operating systems may be required.

Software Fault-Tolerance -- Efforts to attain software that can tolerate software design faults (programming errors) have made use of static and dynamic

redundancy approaches similar to those used for hardware faults. One such approach, *N*-version programming, uses static redundancy in the form of independently written programs (versions) that perform the same functions, and their outputs are voted at special checkpoints. Here, of course, the data being voted may not be exactly the same, and a criterion must be used to identify and reject faulty versions and to determine a consistent value (through inexact voting) that all good versions can use. An alternative dynamic approach is based on the concept of recovery blocks. Programs are partitioned into blocks and acceptance tests are executed after each block. If an acceptance test fails, a redundant code block is executed.

An approach called *design diversity* combines hardware and software fault-tolerance by implementing a fault-tolerant computer system using different hardware *and* software in redundant channels. Each channel is designed to provide the same function, and a method is provided to identify if one channel deviates unacceptably from the others. The goal is to tolerate both hardware and software design faults. This is a very expensive technique, but it is used in very critical aircraft control applications.

A system can be designed to be fault tolerant in two ways[2]. A system may mask failures or a system may exhibit a well-defined failure behavior in the event of failure. When a system is designed to mask failure, it continues to perform its specified function in the event of a failure. A system designed for well-defined behavior may or may not perform

the specified function in the event of a failure, however, it can facilitate actions suitable for recovery.

Issues:

Since a fault-tolerant system must behave in a specified manner in the event of a failure, it is important to study the implications of certain types of failure.

Pprocess deaths. When a process dies, it is important that the resources allocated to that process are recouped, otherwise they may be permanently lost. Many distributed systems are structured along the client-server model in which a client requests a service by sending a message to a server. If the server process fails, it is necessary that the client machine be informed so that the client process, waiting for a reply can be unlocked to take suitable action. Likewise, if a client process dies after sending a request to a server, it is imperative that the server be informed that the client process no longer exists. This will facilitate the server in reclaiming any resources it has allocated to client process.

Machine failure.In the case of machine failure, all the process running at the machine will die. As far as the behavior of a client process or a server process is concerned, there is not much difference in their behavior in the event of a machine failure or a process death. The only difference lies in how the failure is detected. In the case of process death, other processes including the kernel remain active. Hence, a message stating that the process has died can be sent to an inquiring process. On the other hand, an absence of any kind of message indicates either process death or a failure due to machine failure.

Network failure. A communication link failure can partition a network into subnets, making it impossible for a machine to communicate with another machine in a different subnet. A process cannot really tell the difference between a machine and a communication link failure, unless the underlying communication network (such as Ethernet), a fault-tolerant design will have to assume that a machine may be operating and processes on that machine are active.

Atomic Commitment Protocols:

Atomic commitment is one of the key functionalities of mobile communication system. The vast majority of such products implement atomic commitment using some variation of 2 Phase Commit (2PC) although 2PC may block under certain conditions. Usually a machine level instruction, which is indivisible, instantaneous, and cannot be interrupted (unless the system fails) corresponds to an atomic operation. However, it is desirable to be able to group such instructions that accomplish a certain task and make the group an atomic operation. Atomic actions extend the concept of atomicity from one machine instruction level to a sequence of instructions or a group of processes that are themselves to be executed atomically [3]. Atomic actions are the basic building blocks in constructing fault tolerant operations. They provide a means to a system designer to specify the process interactions that are to be prevented to maintain the integrity of the system [4].

A action is atomic [3,4]

- If the process performing it is not aware of the existence of any other active processes, and no other process is aware of the activity of the process during the time the process performs the action.

- If the process performing it does not communicate with other processes while the action is being performed.
- If the process performing it can detect no state change except those performed by itself, and it does not reveal its state changes until the action is complete.

1. Low latency Commit Protocol:

1.) Static Two-Phase Commit Protocol:

The two-phase commit protocol is a distributed algorithm that lets all nodes in a distributed system agree to commit a transaction. The protocol results in either all nodes committing the transaction or aborting, even in the case of network failures or node failures. However, the protocol will not handle more than one random site failure at a time [5]. The two phases of the algorithms are the commit-request phase, in which the coordinators attempts to prepare all the cohorts, and the commit phase, in which the coordinator completes the transactions at all cohorts.

Assumptions: This protocol assumes that one of the cooperating processes acts as a coordinator. Other processes are referred to as cohorts. (Cohorts are assumed to be executing at different sites) This protocol assumes that a stable storage is available at each site and the write ahead log protocol is active. At the beginning of the transaction, the coordinator sends a start transaction message to every cohort.

Phase 1. At the coordinator:

1. The coordinator sends a COMMIT-REQUEST message to every cohort requesting the cohorts to commit.

2. The coordinator waits for replies from all the cohorts.

At cohorts:

1. On receiving the COMMIT-REQUEST message, a cohort takes the following actions.

If the transaction executing at the cohort is successful, it writes UNDO and REDO log

on the stable storage and sends an AGREED message to the coordinator. Otherwise, it

sends as ABORT message to the coordinator.

Phase II. *At the coordinator:*

1. If all the cohorts reply AGREED and the coordinator also agrees, then the coordinator

writes a COMMIT record into the log. Then it sends a COMMIT message to all the

cohorts. Otherwise, the coordinator sends an ABORT message to all the cohorts.

2. The coordinator then waits for acknowledgments from each cohort.

3. If an acknowledgment is not received from any cohort within a timeout period, the

coordinator resends the commit/abort message to that cohort.

4. If all the acknowledgments are received, the coordinator writes a COMPLETE record

to the log (to indicate the completion of the transaction)

At cohorts:

1 On receiving a COMMIT message, a cohort releases all the resources and locks

held by it for executing the transaction, and sends an acknowledgments

2 On receiving an ABORT message, a cohort undoes the transaction using the UNDO log record, releases all the resources and locks held by it for performing the transaction, and sends an acknowledgment.

When there are no failures or message losses, it is easy to see that all sites will commit only when all the participants (Including the coordinator) agree to commit. In the case of lost messages (sent from either cohorts or the coordinator) the coordinator simply resends messages after the timeout. Now we shall attempt to show that this protocol results in all participants either committing or aborting, even in the case of site failures.

2. Dynamic two-phase commit protocol:

A common variant of 2PC in a distributed system, which better utilizes the underlying communication infrastructure, is the *Tree 2PC protocol*. In this variant the coordinator is the root ("top") of a communication tree (inverted tree), while the cohorts are the other nodes. Messages from the coordinator are propagated "down" the tree, while messages to the coordinator are "collected" by a cohort from all the cohorts below it, before it sends the appropriate message "up" the tree (except an abort message, which is propagated "up" immediately upon receiving it, or if this cohort decided to abort).

The Dynamic two-phase commitment protocol is a variant of Tree 2PC with no predetermined coordinator. Agreement messages start to propagate from all the leaves, each leaf when completed its tasks

on behalf of the transaction (becoming *ready*), and the coordinator is determined dynamically by racing agreement messages, at the place where they collide. They collide either on a transaction tree node, or on an edge. In the latter case one of the two edge's nodes is elected as a coordinator (any node). D2PC is time optimal (among all the instances of a specific transaction tree, and any specific Tree 2PC protocol implementation; all instances have the same tree; each instance has a different node as coordinator): it commits the coordinator and each cohort in minimum possible time, allowing earlier release of locked resources. Disadvantages: The greatest disadvantage of the two phase commit protocol is the fact that it is a blocking protocol. A node will block while it is waiting for a message. This means that other processes competing for resource locks held by the blocked processes will have to wait for the locks to be released. A single node will continue to wait even if all other sites have failed. If the coordinator fails permanently, some cohorts will never resolve their transactions. This has the effect that resources are tied up forever.

The greatest disadvantage of the two-phase commit protocol is the fact that it is a blocking protocol. A node will block while it is waiting for a message. This means that other processes competing for resource locks held by the blocked processes will have to wait for the locks to be released.

Non Blocking Commit Protocols:

We present some observation that lead to the conditions under which the two phase commit protocol blocks[6]. Consider a simple case where only one site remains

operational and all other sites have failed. This site has to proceed based solely on its local state. Let's denote the state of the site at this point. If $C(s)$ contains both commit and abort sites, then the site cannot decide to abort the transaction because some other site may be in the commit state. On the other hand, the site cannot decide to commit the transaction because some other site may be in the abort state.

Voting Protocols: A common approach to provide fault tolerance in distributed mobile system is by replicating data at many sites. If a site is not available, the data can still be obtained from copies at other sites. Commit protocols can be employed to update multiple copies of data. While the nonblocking protocol of the previous section can tolerate single site failure, it is not resilient to multiple site failures, communication failures, and network partitioning. In commit protocols, when a site is unreachable, the coordinator sends messages repeatedly and eventually may decide to abort the transaction, thereby denying access to data. However, it is desirable that the sites continue to operate even when other sites have crashed, or at least one partition should continue to operate after the system has been partitioned. Another well known technique used to manage replicated data is the voting mechanism. With the voting mechanism, each replica is assigned some number of votes, and a majority of votes must be collected from a process before it can access a replica. The voting mechanism is more fault tolerant than a commit protocol in that without compromising the integrity of the data. There are two voting mechanisms 1) Static Voting 2) Dynamic Voting.

System Model: The static voting scheme is proposed by Gifford[7]. The replicas of

files are stored at different sites. Every file access operation requires that an appropriate lock is obtained. The lock granting rule allow either 'one writer and no reader' or 'multiple readers and no writers' or 'multiple readers and no writers' to access a file simultaneously . it assumes that at every site there is a lock manager that performs the lock related operation, and every file is associated with a version number, which gives the number of times the file has been updated. The version numbers are stored on stable storage, and every successful write operation on a replica updates its version number.

Basic idea. The essence of a voting algorithm which controls access to replicated data is as follows. Every replica is assigned a certain number of votes. This information is stored on stable storage. A read or write operation is permitted if a certain number of votes, read quorum or write quorum, respectively, are collected by the requesting process.

Each of the (correct) voters will follow the steps below:

1. If no other voter has committed an answer to the interface module yet, the voter does so with its own vote; it then skips the remaining steps.
2. In the case that another voter has committed, the voter compares the committed value from the other voter with its own vote.
3. If the results agree, the voter does nothing; otherwise it broadcasts its dissenting vote to all the other voters
4. Once all voters have had a chance to compare their votes with the committed value (this interval could be determined by a timer),

the voter analyzes all the dissenting votes to determine if a majority dissenting vote exists.

5. If no majority exists, then the voter does nothing.
6. If a new majority exists (or if another, perhaps faulty, voter commits a new result), then the voter returns to step 1.

The interface module will follow these steps:

1. Once a commit is received, the result is stored in the buffer and the timer is started. The timer is set to allow time for all the voters to check the committed value and dissent if necessary.
2. If a new commit is received before the timer runs out, the new result is written over the old one in the buffer, and the timer is restarted.
3. If no commit occurs before the timer runs out, then the interface module sends the result in its buffer to the user, and the algorithm is terminated.

The Voting Algorithm: When a process executing at site i issue a read or write request for a file, the following protocol is initiated.

1. Site i issues a Lock-Request to its local lock manager.
2. When the lock request is granted, site i sends a Vote_Request message to all the sites.
3. When a site j receives a Vote_Request message, it issues a Lock_Request to its local lock manager. If the lock request is granted, then it returns the version number of the replica (VN_j) and

- the number of votes assigned to the replica(V_j) to site i .
4. Site I decides whether it has the quorum or not, based on the replies received within a timeout period as follows(P denotes the set of sites which have replied).

If the request issued was a read,

$$V_r = \sum_{K \in P} V_k$$

$$K \in P$$

If $V_r \geq r$, where r is the read quorum, then site I has succeeded in obtaining the read quorum. If the request issued was a write,

$$V_w = \sum_{K \in Q} V_k$$

$$K \in Q$$

Where the set of sites Q is determined as follows:

$$M = \max\{V_{N_j} : j \text{ belong to } P\}$$

$$Q = \{j \text{ belong to } P : V_{N_j} = M\}$$

In other words, the largest version number M denotes the version number of the current copy, and only the votes of the current replicas are counted in deciding the write quorum. If $V_w \geq w$, where w is the write quorum, then site i has succeeded in obtaining the write quorum.

5. If site i is not successful in obtaining the quorum, then it issues a Release_Lock to the local lock manager as well as to all the sites in P from whom it has received votes.

6. If site i is successful in obtaining the quorum, then it checks whether its copy of the file is current. A copy is current if its version number is equal to M . If the copy is not current, a current copy is obtained from a site that has a current copy. Once a current copy is available locally, site i performs the next step.
7. If the request is a read, site i reads the current copy available locally. If the request is a write, site i updates the local copy. Once all the accesses to the copy are performed, site i updates V_{N_i} , and sends all the updates and V_{N_i} to all the sites in Q .

Reference:

1. B.R. Badrinath, A. Acharya, and T. Imielinski, "Structuring Distributed Algorithms for Mobile Host," Proc. 14th Int'l Conf. Distributed Computing Systems, June 1994.
2. Cristian, F. "Understanding Fault-Tolerant Distributed System," Communication of the ACM, vol. 34, no. 2, Feb. 1991, pp. 56-78.
3. Lomet, D.B., "Process Structuring, Synchronization, and Recovery Using Atomic Actions," Proceeding of the ACM Conference on language Design for Reliable Software, SIGPLAN, Notices 12,3, March 1997, pp. 128-137.
4. Randell, B., "Reliable Computing Systems," Operating System: An Advance Course, Springer-Verlag New York, 1979, pp. 282-391.
5. Skeen, D. (May 1983). "A Formal Model of Crash Recovery in a Distributed System". IEEE transactions on Software Engineering 9(3) pp. 219-228.
6. Skeen, D., "Non A Formal Model of Crash Recovery in a Distributed System,"

IEEE Transactions on Software Engineering, vol. 9, no. 3, May 1983, pp. 219-228.

7. Gifford, D.K., "Weighed Voting for Replicated Data", Proceeding of the 7th ACM Symposium on Operating System Principles, Dec. 1979, pp. 150-162.