An Architecture for Rapid Distributed Fault Tolerance [1]

Dr. Samuel H. Russ
Mississippi State University
NSF Engineering Research Center for Computational Field Simulation

Abstract — Embedded high performance computing is being called upon to provide critical computing resources with increasing frequency. The ability to tolerate faults during operation, both maintaining operational capability and ensuring that correct results continue to be produced, is an important ingredient in mission–critical systems. An architecture for such a system is proposed, providing the ability to withstand faults with graceful degradation in performance and complete transparency to the applications programmer. The final system will be able to offer fault–tolerant computing transparently to MPI applications and draws heavily on existing, demonstrated successes.

1 Introduction and Background

Fault–tolerant computing is a difficult and necessary ingredient in applications that demand high reliability. Compounding the problem is the ready availability of inexpensive but unreliable off–the–shelf hardware and software. Combining unreliable systems to produce a reliable one can be difficult and expensive.

In general, fault–tolerant systems have the means of detecting, correcting, and tolerating faults. The difficulties associated with doing this depend on the reliability of the underlying hardware and error detection mechanism, the availability of correctly operating replacements, the ease with which processing can be transferred, and the structure of the software that is being executed.

In this work we will assume that the application is correctly designed and deterministic. These assumptions are warranted in this context because most embedded applications use well–understood algorithms and because tolerance of flaws in an application's fundamental design is outside the scope of this work.

Based on these assumptions, tolerance of hardware failures can be created by duplicating tasks and the messages that are transferred between them. This type of fault tolerance relies on two sets of criteria in order to work correctly. First, tasks must be duplicated, implying two constraints must be satisfied. Second, messages must be duplicated and processed correctly, entailing the satisfaction of three constraints.

Some explanation of terminology is in order. A *program* or *job* is a software entity to perform computation. It may be parallel, in which case it is considered as a collection of communicating sequential programs called *tasks* or *processes.*

Requirements for task duplication are that a task's state be duplicated *correctly* and *consistently* [1]. Not only must all elements of its state be transferred intact (correctness), but also the state image must be consistent with that of the states of all other tasks (consistency). To put it briefly, the task must "agree" with all other tasks on the status of message delivery before task duplication can proceed. If a message is still in transit when a task is duplicated, the message will not be delivered to the duplicate task and the state of the two tasks will be different.

There are three requirements for proper message delivery (also called "group communication" in this context)—order, atomicity, and termination [2]. First, all message streams must arrive in the same order at each task. This may seem obvious, but becomes problematic in the presence of faults. Second, each message must be delivered. Third, all duplicated tasks must eventually send the same messages. That is, identical copies of

[1] This work was funded in part by NSF Grant No. EEC–8907070 Amendment 021 and by ONR Grant No. N00014–97–1–0116.

the same task must produce the same output. This requires that algorithms be deterministic and that tasks must behave similarly (and not "stall").

2 Using Task and Message Duplication for Fault Tolerance

2.1 Rapid Fault Tolerance

Rapid fault tolerance can be obtained by simply duplicating tasks and having tasks process the first instance of a message that arrives. Duplicate messages may be discarded. The first instance of a collection of duplicated tasks delivering a message is sufficient for the following tasks to continue. Failure of a task, assumed in the context of rapid fault tolerance to be catastrophic, is simply and effectively ignored.

Requiring that tasks fail catastrophically, as opposed to permitting tasks to correctly emit incorrect results, is usually not a stringent requirement. Most modern computer systems effectively exhibit total shutdown in the presence of almost all serious faults, and will likely not "limp along" and produce mathematically incorrect answers. A computer is likelier to fail completely than to conclude that 2+2=5.

One cause of this type of error is single–event upset (SEU) caused by radiation (including naturally occurring radiation inside chip packages). Radiation–tolerant design practices and error–control coding can prevent most SEU events from causing failure. Bad memory chips can also exhibit similar symptoms, and SECDED coding (single–error correction, double–error detection) can correctly indicate this type of failure.

Another cause is a previously undiscovered timing error. This can be caused by a faulty part, operation near or past rated temperature or voltage ranges, or design flaw. One notorious example of this is Intel's original Pentium chip. Designers often use parity bits as flags for timing issues.

2.2 More Reliable Fault Tolerance

In some circumstances, partial failure must be guarded against. Besides the possibility of an SEU or previously undiscovered timing problem, a maliciously altered program will produce incorrect results. There are several ways to protect against it, such as the use of checksums and/or voting logic, and they involve the use of and comparison of results among duplicate tasks. In this case, execution can only continue once enough votes have been gathered and a majority response determined.

This slows down throughput two ways. First, tasks must wait for all duplicates (or nearly all) to progress to the point of message transmission. Second, considerable time may be needed to determine if a majority exists and what the "majority answer" is. Checksums may be one way to speed this determination, but maliciously altered tasks may emit incorrect answers with correct checksums. The malicious case is likelier than the partial–hardware–failure case (if cautious, SEU–resistant design practices are used) and so must be guarded against.

2.3 Bounds on Degrees of Duplication, Fault Tolerance, and Throughput

If a system must tolerate n catastrophic failures, then it requires n+1 duplicates of every task in order to continue. To tolerate m partial failures, 2m+1 votes must be collected. In general, then, to tolerate n catastrophic failures and m partial ones, n + 2m + 1 duplicates are needed. Voting is conducted once 2m+1 out of n+2m+1 votes are gathered.

If all tasks are assumed to fail catastrophically, then the time to issue a message and permit subsequent computation is the minimum completion time over all duplicates. If tasks can fail partially, then the message–issue time is the minimum time to receive m+1 correct messages plus the time to perform the comparison. It is therefore the maximum time over the first m+1 correct messages to arrive.

In general, then, to survive n catastrophic failures and m partial failures, a system must have

$$N_{dup} = n + 2m + 1 \tag{1}$$

where N_{dup} is the number of duplicates of every task. The time to proceed is

$$T_{proc} = \max_{m+1 correct \backslash N_{dup}} (\min[t_{proc_i}]) + T_{comp} \quad (2)$$

where t_{proci} is the time for task i to send the message, T_{comp} is the time to compare messages, and T_{proc} is the overall time to receive a message. The min term represents the fastest tasks to complete message transmission over all N_{dup} duplicates. The max term represents the last needed correct message, which is (m+1)st correct message.

Catastrophic failure (m=0) actually requires substantially less task duplication and downstream processing to compensate. For example, the max expression in equation (2) simplifies to the maximum time over 1 task, and thus is actually the minimum time over all duplicates.

2.4 Other Applications of the Technique

Security

In order for sabotage to succeed, a majority of tasks must be simultaneously corrupted. If they are not corrupted at the same time, tasks corrupted earlier will be outvoted and purged, leaving later tasks to be outvoted and purged as well.

Speedup

The ability to duplicate tasks and messages opens up the possibility of schedulers with "speculative scheduling". Duplicates of tasks may finish earlier if the machine on which they run is faster. This speedup can have nearly instantaneous effects on other tasks, and so the overhead of duplicated messages may be offset by the substantial time savings. This assumes, of course, that sufficient extra processing hardware is present to support the duplicated tasks.

3 Software Infrastructure and Accomplishments to Date

3.1 Task Duplication

As outlined above, tasks must be duplicated correctly and consistently. This can be accomplished either by having the programmer write checkpointing routines or by developing a method for a program to transfer its own state to a duplicate of itself. There are several advantages and disadvantages of each approach. If the user writes a checkpointing routine, the routine is typically able to be invoked at fixed points in a program's execution. Thus the user may be able to take advantage of internal knowledge of a program's structure in order to select points where the program state is relatively small. Further, if the checkpoint format is architecture–independent, this permits tasks to be migrated across various architectures. If the checkpointing is provided by the run–time system, it is available at any point in the program's execution and can be provided transparently to programs running under it. Migration across architectures is not generally possible however. For example, pointers are not the same and variables stored in registers in one architecture may be stored in physical memory in another.

Imposing the requirement that fault–duplication be possible on demand makes the design of a complete system easier, as duplication can proceed immediately upon the detection of a fault so that the required number of duplicates can be maintained. Fault recovery, then, simply becomes a matter of having surplus processing capability and the ability to run newly created duplicates on the surplus hardware. This means that on–demand duplication is preferred, and therefore user–transparent, homogeneous–platform migration is the preferred migration method in this environment.

The Hector system, under development at Mississippi State, has already demonstrated the ability to duplicate tasks correctly, consistently, and immediately [3]. The state–transfer method that Hector uses may be invoked at any point in a program's execution, and so tasks are always available for duplication. Additionally, because Hector uses a specially modified MPI implementation, this task duplication is available for unmodified MPI

programs. This method has been used to migrate tasks in mid–computation to other idle resources, and can easily be adapted to provide task duplication[4].

The correctness criterion is satisfied by transferring all elements of the process's running state. (Note that we are referring to a task using the terminology of the operating system. Each task is a Unix process.) This is non–trivial, as the process's kernel keeps part of the program's state, such as file descriptors, file pointers, and signal handler status. This information is tracked using "wrapper functions" so that important details of the state maintained by the kernel are also kept in the process's address space. The consistency criterion is satisfied by having tasks exchange "end–of–channel" messages so as to insure that no messages are lost in transit [3].

3.2 Correct Message Delivery

Messages must be delivered subject to the three constraints listed above (order, atomicity, and termination). There are additional, implicit constraints that each task must be able to send, receive, and appropriately process duplicate messages.

Additional features, such as unique message identifiers, may prove helpful in ordering incoming messages. Another capability, to be able to rollback the state of a task to before a message was received, would prove helpful if partial failures must be tolerated, but may be expensive if the program's state is large.

The Isis communications library is widely regarded as the first to offer message duplication services [2]. Because both it and its "offspring" Horus are implemented in a layered fashion, it can be adopted by other programs or libraries that need duplication. The extra message duplication layer can handle the issues associated with duplication, decimation (elimination of unwanted duplicates at the receiving end), and voting logic.

3.3 Programmer Transparency

It would be highly desirable for such a system to run parallel jobs transparently. That is, it would be desirable if the programmer did not have to rewrite or add to an application in order to run it in a fault–tolerant environment. This can be supported by using a library approach to expressing parallelism and inter–process communications [5]. As long as the programmer wrote the parallel program using the library, all of the features listed above could be provided automatically through a modified library. This is the approach used by Hector [5]. Its modified MPI library provides for task duplication and performance instrumentation, and can be extended to support group communications.

3.4 Control System

Some sort of control system is needed to detect faults, order task duplication, and purge tasks suspected of being faulty. Ideally this control system should itself be fault–tolerant. The control system may be called on to perform other duties. Additional synchronization is needed, for example, during task duplication in order to ensure that all end–of–channel messages have been exchanged.

The Hector system already runs in a distributed manner [5]. Each platform has running on it a small utility task (a "slave allocator") that monitors program execution and platform loading. Performance measurements are made periodically and sent to a central optimization facility. Because these measurements are periodic, rapid detection of catastrophic failure is possible and could be used to invoke task duplication. Detection of a partially (or maliciously) failed task is more difficult, requiring the voting logic software to report failures and a mechanism to systematically purge faulty tasks.

A complete system is illustrated below in Fig. 1.

4 Future Work

4.1 Integration of Message Duplication

The Hector system can duplicate tasks already. Once a duplicate task is created, tasks that send messages to the task must also send them to the duplicate. Likewise, tasks that

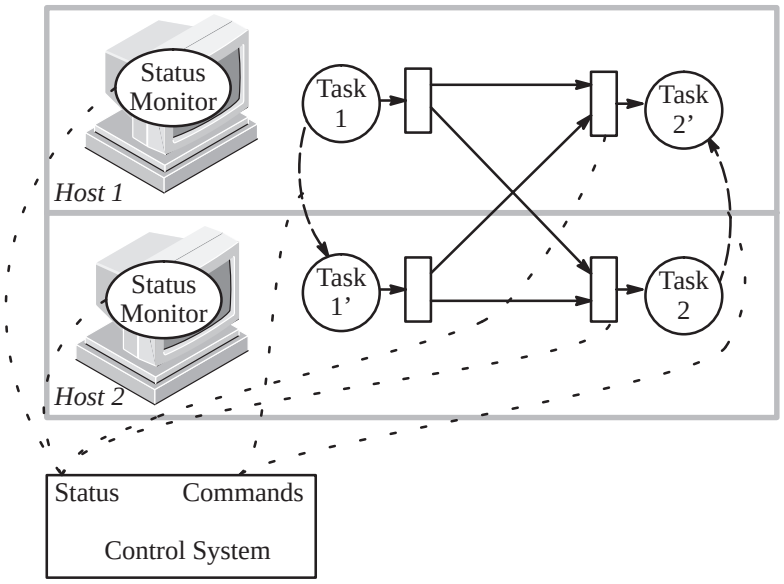

Fig. 1. A Complete Fault–Tolerant Run–Time System

receive messages from the duplicated tasks must process duplicate messages appropriately. Exploration of existing message duplication libraries will be made, as one may be suitable for adoption by Hector. (See [2] for an example list.) Hector's migration protocol already interrupts the communication at a consistent point, and so extension of the protocol for duplication can easily be made.

One modification will need to be made to the message–duplication library. Duplicate tasks can appear and disappear at any time, and so delivery of messages will have to be structured to permit dynamic creation of duplicate message streams. This will eliminate the need for some sort of rollback strategy to process or deliver messages lost between the fault and the creation of a new duplicate.

4.2 Distributed Fault–Tolerant Control

Hector is able to detect catastrophic failures by having a "time–out" for lost machine status updates. This can be added fairly easily, as these status updates are made periodically under normal circumstances. An interface into tasks' voting logic will be needed if detection of partially failed tasks is desired.

Hector's current centralized control system is currently a single point of failure, as the central optimization facility is not duplicated. Design alternatives include using the message–duplication approach to duplicate the central optimizer, having the distributed utility tasks (slave allocators) take over the optimization responsibility, and having tasks monitor the machine on which they run and order their own migrations.

5 Bibliography

[1] Samuel H. Russ, Brian Flachs, Jonathan Robinson, and Bjorn Heckel, ''Hector: Automated Task Allocation for MPI'', *Proceedings of the 10th International Parallel Processing Symposium*, Honolulu, HI, April 1996.

[2] Guerraoui, R., and Schiper, A., "Software–based Replication for Fault Tolerance", *Computer*, Vol. 30, No. 4, April 1997, pp. 68–74.

[3] Jonathan Robinson, Samuel H. Russ, Brian Flachs, and Bjorn Heckel, ‘‘A Task Migration Implementation for the Message–Passing Interface’’, *Proceedings of the IEEE 5th High Performance Distributed Computing Conference (HPDC–5)*, Syracuse, NY, August 1996.

[4] Dr. Samuel H. Russ, “Using Hector in an Architecture for Rapid Distributed Fault Tolerance”, *MSU Technical Report No. MSSU–EIRS–ERC–97–17*, December 1997.

[5] Dr. Samuel H. Russ, Brad Meyers, Chun–Heong Tan, and Bjorn Heckel, “User–Transparent Run–time Performance Optimization”, *2nd International Workshop on Embedded High–Performance Computing*, associated with IPPS ’97, Geneva, Switzerland, April 1997.