Tools and Techniques For Adding Fault Tolerance to Distributed and Parallel Programs*

Ozalp Babaoglu**

TR 91-1249
December 1991

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

---

*This work was supported in part by by the Commission of European Communities under ESPRIT Programme Basic Research Action Number 3092 (Predictably Dependable Computing Systems), the United States Office of Naval Research under contract N00014-91-J-1219, IBM Corporation and the Italian Ministry of University, Research and Technology.
** On leave from the Department of Mathematics, University of Bologna, 40127 Bologna, Italy.

Tools and Techniques for Adding Fault Tolerance to Distributed and Parallel Programs*

Özalp Babaoğlu†

Department of Computer Science
Cornell University
Ithaca, New York 14853-7501

December 7, 1991

Abstract

The scale of parallel computing systems is rapidly approaching dimensions where fault tolerance can no longer be ignored. No matter how reliable the individual components may be, the complexity of these systems results in a significant probability of failure during lengthy computations. In the case of distributed memory multiprocessors, fault tolerance techniques developed for distributed operating systems and applications can be applied also to parallel computations. In this paper we survey some of the principal paradigms for fault-tolerant distributed computing and discuss their relevance to parallel processing. One particular technique — passive replication — is explored in detail as it forms the basis for fault tolerance in the Paralex parallel programming environment.

Keywords: Parallel processing, reliability, transactions, checkpointing, recovery, replication, reliable broadcast, causal ordering, Paralex.

*This work was supported in part by the Commission of European Communities under ESPRIT Programme Basic Research Action Number 3092 (Predictably Dependable Computing Systems), the United States Office of Naval Research under contract N00014-91-J-1219, IBM Corporation and the Italian Ministry of University, Research and Technology.

†On leave from the Department of Mathematics, University of Bologna, 40127 Bologna, Italy

1 Introduction

Until recently, designers of parallel scientific programs have included little or no support for fault tolerance in their applications. This attitude has been justified through the follows observations: (i) the modest size of existing multiprocessor hardware platforms have made failures relatively rare events, (ii) programming fault-tolerant applications has meant mastering complex distributed computing concepts, (iii) the overhead for fault tolerance has been judged to be too high with respect to desired performance.

With the advent of massively-parallel machines with tens of thousands of processors and complex interconnection networks (e.g., the Connection Machine [20], the J-Machine [18]), application-level fault tolerance support has to be reconsidered. In machines of this size, hardware-based fault tolerance, such as those employed in the Tandem [7] and Stratus [35] systems, is clearly impractical. No matter how reliable the individual components are, the sheer size of these systems can result in a significant probability of failure during lengthy computations. If parallel applications that use large numbers of processors are to make progress, they have to anticipate the possibility of partial failures and take appropriate steps to recover from them. Unless this is done, reliability will become the limiting factor in the parallelism that can be achieved by applications [29].

Shared-memory multiprocessors present severe architectural problems when scaled to very large dimensions. It is widely accepted that constructing parallel machines that can scale to very large numbers of processors will be possible only for distributed-memory architectures. Physical properties of these machines will prevent relying on a global clock as a time base and partial failures will result in loss of communication or computation without bringing down the entire system. In other words, the loose coupling that is dictated by size will render these machines equivalent to "distributed systems in a box."

Extremely fast networking is yet another trend that supports this "distributed system" view of parallel multiprocessors. With the possibility of Gigabits-per-second communication over large geographic distances [21], an entire network of machines (parallel or scalar) can be thought of as a parallel multiprocessor. Existing efforts in the United States linking distant supercomputing centers across the country with high-speed communication lines support this observation. Even on a more modest scale, there are many efforts to support parallel computing over networks (Ethernet LANs) of workstations [5, 8].

The realization that parallel multiprocessors are logically (or physically, as in the case of network-based computing) equivalent to distributed systems has two consequences. First, fault-tolerant parallel computing in distributed memory multiprocessors has to be solved in the presence of uncertainties that are inherent to distributed systems. Second, the wide body knowledge that has been accumulated for fault-tolerant distributed computing can be directly applied to fault-tolerant parallel computing.

The remainder of this paper is organized as follows. In the next section we present a brief survey of the major paradigms for fault-tolerant distributed computing. Transactions, checkpointing, active replication and passive replication are examined and evaluated as possible mechanisms for fault-tolerant parallel computing. Section 3 is a brief introduction to the ISIS distributed programming toolkit that includes the necessary primitives for implementing a wide range of fault tolerance mechanisms. Section 4 is an overview of the Paralex programming environment that permits parallel applications to be developed and executed in distributed systems with automatic support for fault tolerance. Use of passive replication to render parallel programs fault tolerant in

Paralex is discussed in Section 5. We conclude the paper by some observations derived from our experience with Paralex.

2 Paradigms for Fault-Tolerant Distributed Computing

Failures in a system can result in the loss of data or computation. In this paper we will be addressing only the possibility of tolerating failed computations. While maintaining data correct and available is an equally important concern, it is beyond the scope of this paper.

All other things being equal, a distributed application will be less reliable than its centralized equivalent — there are simply more components that the distributed application depends upon and each can fail independently. For distributed systems to be useful, they have to be fault tolerant.

Tolerating failures in any system requires some form of redundancy. In *time redundancy*, the failed computation is restarted on the same processor (once the cause of the failure has been eliminated) or on another processor and repeated until it completes successfully. In *space redundancy*, the computation is carried out on several physically independent processors in parallel and a vote is taken to extract a single output from the (potentially different) results.

It is clear that space-redundant systems are more expensive in terms of computational resources. In return, they are able to mask out failures and continue producing correct outputs with no loss in performance[1]. This makes space-redundant systems suitable for time-critical applications such as process control. Time-redundant systems, on the other hand, go through a recovery phase where no useful computation is being carried out. They also require the ability to detect failures before (incorrect) results are communicated externally. Parallel scientific applications typically do not have critical timing constraints to justify the cost of space redundancy. If, however, the parallelism available in the hardware exceeds that usable by the application, the extra processors may be put to good use by running replicas of the primary computation.

Achieving fault tolerance through redundancy in distributed systems requires that computations on different processors cooperate. The lack of shared memory and the lack of a global clock makes reasoning about such systems a difficult task. Since message exchange is the only means of communication and it incurs random delays, it is impossible for any one component to have an instantaneous view of the global computation state. The possibility of processor and communication failures further increases the level of uncertainty in these systems and adds to their conceptual difficulty. We can hope to master fault-tolerant distributed computing only through the use of appropriate paradigms that abstract away many of these complexities [30, 16]. In the following sections we present some of these paradigms.

2.1 Transactions

Transactions were originally proposed as a software structuring mechanism for applications that accessed shared data on secondary storage (typically a database) [19]. In this model, computations are divided into units of work called *transactions*. The system guarantees three properties for transactions: atomicity, serializability and permanence. Atomicity is with respect to failures in the sense that the execution of a transaction is "all or nothing" — failures never leave intermediate states of a transaction visible to other transactions. Serializability, on the other hand, requires that

[1] As we shall see, there is a slight degradation in performance due to the dissemination of inputs to the replicas and due to the voting at the output.

the effect of concurrent execution of several transactions be equivalent to some serial execution (one after the other in some arbitrary order). Permanence guarantees that computations make progress despite failures since their results will never be undone.

Programming with transactions presents to the user an idealized world where failures and other concurrent transactions have been abstracted away. The system automatically restarts transactions if a failure interrupts their execution part way or if serializability cannot be guaranteed. Once a transaction commits, it can be sure that the data values written are as if it executed in isolation and without any failures. Thus, transactions transform the system from one consistent state to another. By definition, transaction boundaries always define consistent system states from which a computation can recover. The basic transaction model has been extended to distributed systems [26].

One of the drawbacks of the transactional model is that fault tolerance cannot be integrated transparently to applications. Programs must explicitly use the transaction paradigm by announcing the beginning and end of transactions within programs at opportune points. Furthermore, while the serializability requirement may be appropriate for database applications, it can be overly restrictive for parallel computations that do not access shared files. The overhead introduced by the complex mechanisms that implement the transaction abstraction may be significant for most parallel applications.

Modern systems that adopt the transaction model as the basis for fault-tolerant distributed computing include Arjuna [32], Argus [27] and Camelot [33].

2.2 Checkpointing

An arbitrary distributed computation could be made fault tolerant without having to structure it as a collection of transactions. All that is required is a mechanism whereby computations can be restarted from some past state in response to failures. To prevent having to restart computations always from the very beginning, and thus guarantee forward progress, the state of the failure-free execution is periodically saved to stable storage[2]. The saved past states are called *checkpoints*. Restoring the system to a set of checkpoints and repeating the lost computations is called *recovery*. The frequency with which checkpoints are taken is a system tuning parameter and establishes the relative costs of the failure-free execution overhead and recovery delays.

In a system where computations interact by exchanging messages, recovery of a failed computation from an arbitrary set of checkpoints may result in an inconsistent global system state [14]. Intuitively, recovery should never be attempted from a system state in which some computation appears to have received messages that have not yet been sent. The manner in which global system state consistency is guaranteed results in two distinct strategies.

2.2.1 Optimistic Recovery

The general strategy is to design algorithms with the guess that failures will not occur at inopportune times. As a recovery strategy, this leads to establishing checkpoints without any coordination among the components. However, the system must have collected sufficient information along the way so that exactly those computations that have to recover do so in case failures occur. For

[2] Stable storage is a memory device whose contents survive all failures short of disasters. It is typically implemented using mirrored disks.

example, in the scheme proposed by Strom and Yemini [34], checkpointing and message logging occur concurrently with computation and communication. Causality information is maintained such that recovery will occur from a consistent global system state. In a variant of the scheme, messages are logged in the nonvolatile memory of the sender rather than the receiver, resulting in even further concurrency of stable storage writes with respect to computations [22]. An unfortunate consequence of optimistic strategies is that recovery time is difficult to bound since, in addition to the failed computation, an arbitrary number of others may need to recover.

2.2.2 Conservative Recovery

A reasonable alternative to the above strategy is to structure the checkpointing mechanism in a manner such that the set of latest checkpoints is always guaranteed to represent a consistent system state. To prevent computations from having to recover arbitrarily past states, conservative schemes synchronize checkpointing with computation and communication. This has the desirable consequence that recovery is both simple and more predictable in the delays it introduces to the system. The cost, obviously, is shifted from recovery to checkpointing.

One way to guarantee consistency of checkpoints is to force each computation to record its state after every message send operation and before doing anything else. Recovery consists of the failed computation rolling back to its most recent checkpoint. This simple mechanism can be extended to cope with missing messages. Unfortunately, this naive solution is impractical since checkpointing to a stable store after every send will introduce significant delays to the computation. We return to this issue in Section 2.4 where we discuss passive replication.

Consistency of checkpoints can be guaranteed even when they are taken much less frequently. Koo and Toueg present a distributed algorithm that guarantees the set of most recent checkpoints to always represent a consistent state [23]. A unilateral checkpoint action forces the minimum number of additional computations to checkpoint along with it. Recovery also involves the minimum number of computations that are affected by the failure.

2.3 Active Replication

Given that a distributed system contains multiple processing elements with independent failure modes, a distributed service can be made more reliable by performing it in parallel on several processors. This simple idea contains numerous subtleties that have to be addressed before it can be made effective.

If a collection of replicas is to be functionally equivalent to a single component, it must accept the same input and produce the same output. Clients of the replicated service continue to interact with it as if it were implemented as a single component. On the client side, code fragments intercept the client request and distribute it to the replicas. On the service side, code fragments intercept an incoming request and engage in communication with all of the replicas of the service to achieve the input dissemination. Finally, the outputs must be coalesced in to a single value. All of this code to wrap around clients and services can be generated automatically using technology similar to Remote Procedure Call (RPC) stub generation [12].

We begin with the problem of coalescing the output. If only benign failures[3] are to be tolerated, then the first output to be produced by some replica can be taken as the component output. To

[3] Benign failures cause components to simply stop and produce no output.

tolerate up to k such failures, it clearly suffices to have $k+1$ replicas. If failures can cause incorrect results to be produced by the replicas[4], then a majority vote will determine the output. This clearly requires $2k+1$ replicas to tolerate up to k failures. It also requires a (reliable) component to act as the voter.

Distributing the input to the replicas is even more subtle. For the above voting scheme to work, all correct replicas must produce the same output. This requires that they all see the same input and that the computations they perform be deterministic. The input must be disseminated such that either all or none of the replicas see it. Protocols that achieve this in the presence of failures are called *reliable broadcast* protocols [15, 3]. If the service interacts with multiple clients, the replicas must not only see the same input, but also see them in the same order. Achieving this in the presence of failures requires the use of an *atomic broadcast* protocol [17]. Depending on the failure assumptions and the system model, achieving atomic broadcast may require $3k+1$ replicas to tolerate up to k failures [25]. Thus, this may be the dominant factor in determining the replication level rather than simple majority.

The above ideas have been expounded in a general methodology called the *state machine approach* for automatically adding fault tolerance to distributed services [31]. It is important to note that while active replication can result in higher reliability of services in the short run, these systems become less reliable than their non-replicated counterparts in the long run [2, 36]. To maintain reliability levels sufficiently high over long intervals, it must be possible to vary the number of replicas dynamically — failed ones must be removed off line and new or repaired ones brought on line. The difficulty in achieving this is maintaining a consistent view of the replica set among the processors. This in turn requires a solution to the *group membership* problem [28].

2.4 Passive Replication

While active replication is able to mask failures without any recovery delays, it is costly — all of the replicas compute actively consuming resources. Unless the system has an abundance of processors, the approach may not be practical. *Passive replication* offers a more economical alternative. The service is replicated just as before, however, only one of the replicas computes while the others remain dormant. If the initial computation reaches completion, no further action is necessary. If a failure prevents the first replica from completing, one of the dormant copies is activated and resumes computing from where it last left off. Thus, in the failure-free scenario, no computation is wasted[5].

Several observations are in order. First, the technique is effective only against benign failures — it is not possible to detect incorrect results. Second, there must be a failure detector so that a passive replica may be started if the initial computation fails. Third, input to the replicas must be disseminated atomically just as in active replication. Finally, the technique incurs a delay while the newly-activated replica "catches up" with the failed computation by processing its input queue.

By far the most common realization of passive replication involves two copies, one known as the *primary* and the other as the *secondary* backup [7, 13]. In this scheme, each communication step requires atomically delivering the message to three destinations: the secondary of the sender and the two copies of the destination. When a secondary takes over upon the failure of the primary, it recovers by processing the messages in its input queue. Having seen the messages sent by the

[4]These types of failures are sometimes called *malicious* or *Byzantine*.

[5]There is a small overhead in keeping the replicas coordinated as discussed later.

primary before it failed serves to prevent the secondary from resending them during recovery. Only when the secondary has reached the state of the primary before it failed, does it engage in active message sending.

While at first sight the primary-secondary replication scheme may seem very different from checkpointing, the two are actually logically equivalent. Consider the checkpointing scheme with conservative recovery where the computation is checkpointed after every send operation. If these synchronous operations are to occur to stable storage, the delays would be intolerable. Rather than representing a checkpoint as a process memory image on disk, we could choose to represent it as a process state on another processor (the secondary) along with a count of sent messages. Replaying the enqueued input messages at the secondary and discarding a number of output messages equal to the primary count effectively restores the secondary state to that of the primary at the point of the last send before the failure. The technique trades off delays in checkpointing (an atomic three-way multicast rather than a write to stable store) with those of recovery (computation rather than restoring the state from stable store). Given that failures a relatively rare in most systems, the approach is very reasonable.

3 The ISIS Distributed Programming Toolkit

From the above discussion, a relatively small number of abstractions have emerged as being necessary for implementing a wide range of fault tolerance paradigms. Furthermore, we have seen that replication plays a fundamental role in achieving fault tolerance. The ISIS toolkit has been designed to facilitate easy construction of efficient distributed programs and to make them fault tolerant [9, 11].

As we discussed in Section 2, the principal difficulty in reasoning about distributed systems is the uncertainty due to communication and failures. Without the appropriate tools, a programmer has to consider an extremely large number of possible executions when developing applications. For example, a message broadcast to a group of processes by simple send operations may be received by some and not received by others. Two concurrent broadcasts to the same set of processes may be received in a different order by some of the members. Events corresponding to processes joining or leaving (either voluntarily or due to a failure) a computation may be perceived by the members in different order with respect to ongoing communication. The ISIS toolkit tries to put order to this complex world. By using the appropriate communication primitives and relying on lower-level support, many of the events in a distributed system can be made to appear as if they occurred at the same instant in all components of a computation. The resulting system, called *virtually synchronous*, offers tremendous intellectual economy to application developers [10].

ISIS runs on a large numbers of systems and extends the basic operating system primitives with the following abstractions:

Process Groups These are the principal structuring constructs for ISIS applications. A process group is a named collection of processes. Process groups may overlap in arbitrary ways to reflect the natural structure of the application. Group membership is dynamic in that processes may join or leave at will. A built-in failure detector turns failures into group departures of the appropriate processes. The group name may be used to address all current members without having to know their individual identities. There are no restrictions on the computation being carried out by the members of a group — they need not be replicas of the same computation.

Group Communication Applications in ISIS are structured as communicating process groups. All data exchanged between groups are encoded as ISIS messages, providing a uniform representation across heterogeneous architectures. ISIS protocols ensure that if a message broadcast to a group is received by one of its members, it is received by all of its members, despite benign processor and communication failures. With respect to ordering, ISIS provides three alternatives:

- **FIFO Broadcast** Only broadcasts originating from the same source are received in the same order by the process group members.
- **Causal Broadcast** Only broadcasts that are causally related are received in the same order. Two broadcasts are said to be *causally related* if there exists a chain of communication events such that one can affect the contents of the other [24]. Unrelated broadcasts may be ordered arbitrarily. ISIS maintains the causality relation even across process group boundaries.
- **Atomic Broadcast** All broadcasts to the group are received in the same order by all of its members. This is true even for broadcasts that are causally unrelated. While the cost of FIFO and Causal broadcasts are comparable, Atomic broadcast incurs a quantitative increase in time delays.

State Transfer To facilitate coordination among group members, ISIS provides a mechanism whereby the state of one member is copied to another. What constitutes the process state is application dependent and is specified by the programmer. State transfers are typically used to initialize the state of a new process joining a group. As with the join event itself, the state transfer is ordered consistently by all group members with respect to communication events.

Given the above abstractions, it is possible to implement almost all of the paradigms of Section 2. The lack of relevant concepts such as serializability and atomic commitment make transactions difficult to implement in ISIS.

Realizing active replication through process groups is immediate. Each computation to be made fault tolerant is replicated to form a process group. All point-to-point communication is replaced with atomic broadcasts to the relevant groups to achieve input dissemination. Since clients may be replicated in addition to servers, each input request may be received multiple times by the members of the server group. Some deterministic function (e.g., majority, mean, median) will have to be applied to the copies of the input to select the value to use. This corresponds to the output voting step of the active replication scheme. Even when the replication level is dynamic, the input extraction function can be implemented by the replicas interrogating the current group membership.

Process groups also form the basis for passive replication. Just before they start computing, all members of a process group invoke the `coordinator-cohort` tool of ISIS which effectively selects one member (the coordinator) to continue computing while the others (cohorts) remain inactive. If ISIS detects the failure of the coordinator before its role comes to completion, it will nominate one of the cohorts to the role of coordinator and resume its execution. While requests are disseminated to group members using atomic broadcast as in active replication, there is no need for a voting (input extraction) function since only one output will be produced (that of the coordinator).

Finally, the state transfer mechanism of ISIS provides a way to implement fault tolerance through checkpointing. State transfers can be requested either to another process or to a disk file. To the extent that a disk approximates stable storage, a failed computation can be resumed from the most recent state found in the file.

4 Parallel Computing in Distributed Systems with Paralex

Paralex is a programming environment for developing parallel applications and executing them on a distributed system, typically a network of workstations. Programs are specified in a graphical notation and Paralex automatically handles distribution, communication, data representation, architectural heterogeneity and fault tolerance. It consists of four logical components: A graphics editor for program specification, a compiler, an executor and a runtime support environment. These components are integrated within a uniform graphical programming environment. Here we give a brief overview of Paralex. Details can be found in [5].

The programming paradigm supported by Paralex is a restricted form of data flow [1]. A Paralex program is composed of *nodes* and *links.* Nodes correspond to computations and the links indicate the flow of (typed) data. Thus, Paralex programs can be thought of as directed graphs (and indeed are visualized as such on the screen) representing the data flow relations plus a collection of ordinary code fragments to indicate the computations. The current prototype limits the structure of the data flow graph to be acyclic.

The semantics associated with this graphical syntax obeys the so-called "strict enabling rule" of data-driven computations in the sense that when all of the links incident at a node contain values, the computation associated with the node starts execution transforming the input data to an output. The computation to be performed by the node must satisfy the "functional" paradigm — multiple inputs, only one output with no side effects. The actual specification of the computation may be done using whatever appropriate notation is available including standard sequential programming languages, parallel programming notations (if the distributed system includes nodes that are themselves multiprocessors), executable binary code or library functions for the relevant architectures.

Unlike classical data flow, the nodes of a Paralex program carry out significant computations. This so-called *large-grain* data flow model [6] is a consequence of the properties of the underlying distributed system where we seek to keep the communication overhead via a high-latency, low-bandwidth network to reasonable levels.

There are many situations where the single output value produced by a node needs to be communicated to multiple destinations as input so as to create parallel computation structures. In Paralex, this is accomplished simply by drawing multiple output links originating from a node towards the various destinations. To economize on network bandwidth, Paralex introduces the notion of *filter* nodes that allow data values to be extracted on a per-destination basis before they are transmitted to the next node. Conceptually, filters are defined and manipulated just as regular nodes and their "computations" are specified through programs. In practice, however, all of the data filtering computations are executed in the context of the single process that produced the data rather than as separate processes to minimize the system overhead.

Once the user has fully specified the Paralex program by drawing the data flow graph and supplying the computations to be carried out by the nodes, the program can be compiled. The first pass of the Paralex compiler is actually a precompiler to generate all of the necessary stubs to wrap around the node computations to achieve data representation independence, remote communication and replica management for those nodes with fault tolerance needs. Type checking across links is also performed in this phase. Currently, Paralex generates all of the stub code as ordinary C. As the next step, the C compiler is invoked to turn each node into an executable module.

The Paralex compiler must also address the two aspects of heterogeneity: data representation and instruction sets. Paralex uses the ISIS toolkit as the infrastructure to realize a universal data representation. All data that is passed from one node to another during the computation are encapsulated as ISIS messages. Heterogeneity with respect to instruction sets is handled by invoking remote compilations on the machines of interest and storing multiple executables for the nodes.

The Paralex executor launches the parallel computation on the distributed system respecting all architectural constraints. Details of how Paralex computation graphs are mapped onto the hosts of a distributed system and how the execution is monitored and controlled dynamically are described in [4].

5 Replication in Paralex

One of the primary characteristics that distinguishes a distributed system from a special-purpose super computer is the possibility of partial failures during computations. These failures may be due to real hardware faults or, more probably, as a consequence of user actions such as rebooting or turning off workstations. To render distributed systems suitable for long-running parallel computations, automatic support for fault tolerance must be provided. The Paralex run-time system contains the primitives necessary to support fault tolerance and dynamic load balancing.

As part of the program definition, Paralex permits the user to specify a fault tolerance level for the computation graph. Paralex will generate all of the necessary code such that when a graph with fault tolerance k is executed, each of its nodes will be executed on $k + 1$ distinct hosts to guarantee success for the computation despite up to k failures. Failures that are tolerated are of the benign type for processors (i.e., all processes running on the processor simply halt) and communication components (i.e., messages may be lost). There is no attempt to guard against more malicious processor failures nor against failures of non-replicated components such as the network interconnect.

Paralex uses passive replication as the basic fault tolerance technique. Given the application domain (parallel scientific computing) and hardware platform (networks of workstations), Paralex favors efficient use of computational resources over short recovery times in its choice of a fault tolerance mechanism. Passive replication not only satisfies this objective, it provides a uniform mechanism for dynamic load balancing through late binding of computations to hosts.

Paralex uses the ISIS *coordinator-cohort* toolkit to implement passive replication. Each node of the computation that requires fault tolerance is instantiated as a process group consisting of replicas for the node. One of the group members is called the *coordinator* in that it will actively compute. The remaining members are *cohorts* and remain inactive other than receiving broadcasts addressed to the group. When ISIS detects the failure of the coordinator, it automatically promotes one of the cohorts to the role of coordinator.

Data flow from one node of a Paralex program to another results in a broadcast from the coordinator at the source group to the destination process group. Only the coordinator of the destination node will compute with the data value while the cohorts simply buffer it in an input queue associated with the link. When the coordinator completes computing, it broadcasts the results to the process groups at next level and signals the cohorts (through another intra-group broadcast) so that they can discard the buffered data item corresponding to the input for the current invocation. Given that Paralex nodes implement pure functions and thus have no internal state, recovery from a failure is trivial — the cohort that is nominated the new coordinator simply starts computing with the data at the head of its input queues.

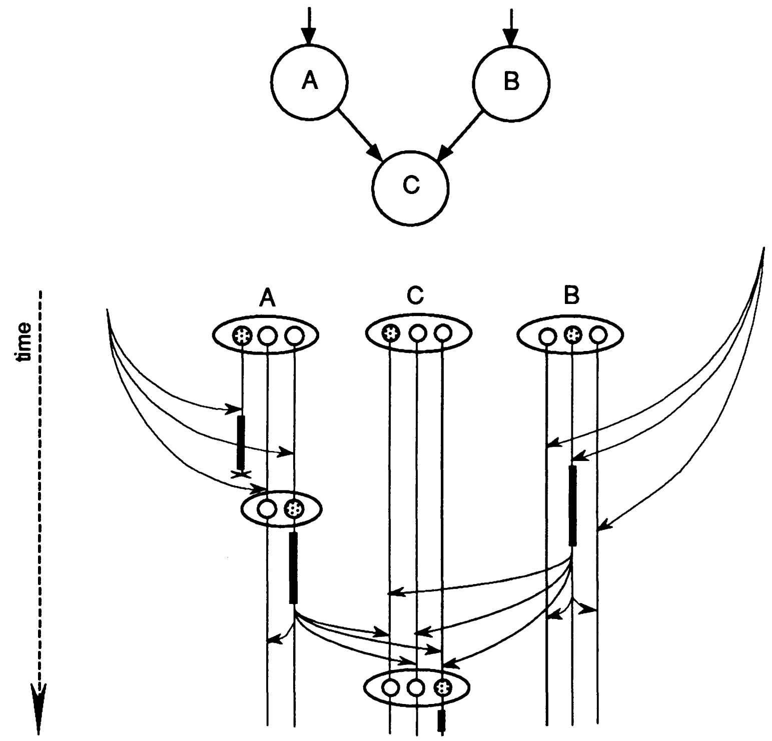

Figure 1: Replication and Group Communication for Fault Tolerance.

Figure 1 illustrates some of these issues by considering a 3-node computation graph shown at the top as an example. The lower part of the figure shows the process group representation of the nodes based on a fault tolerance specification of 2. Arrows indicate message arrivals with time running down vertically. The gray process in each group denotes the current coordinator. Note that in the case of node A, the initial coordinator fails during its computation (indicated by the X). The process group is reformed and the right replica takes over as coordinator. At the end of its execution, the coordinator performs two broadcasts. The first serves to communicate the results of the computation to the process group implementing node C and the second is an internal group broadcast. The cohorts use the message of this internal broadcast to conclude that the current buffered input will not be needed since the coordinator successfully computed with it. Note that there is a small chance the coordinator will fail after broadcasting the results to the next node but before having informed the cohorts. The result of this scenario would be multiple executions of a node with the same (logical) input. This is easily prevented by tagging each message with an iteration number and ignoring any input messages with duplicate iteration numbers.

The execution depicted in Figure 1 may appear deceptively simple and orderly. In a distributed system, other executions with inopportune node failures, message losses and event orderings may be equally possible. What simplifies the Paralex run-time system immensely is structuring it on top of ISIS that guarantees "virtual synchrony" with respect to message delivery and other asynchronous events such as failures and group membership changes. Paralex cooperates with ISIS toward this goal by using a reliable broadcast communication primitive that respects causality [24].

6 Conclusions

We have argued that current large-scale parallel multiprocessors have properties not unlike distributed systems. With expected increases in the scale of parallel machines and increases in network bandwidth of distributed systems, the distinction between them is rapidly fading. This leads us to conclude that future parallel applications will have to confront fault tolerance just as current distributed systems have to. Furthermore, the same tools and techniques to render distributed systems fault tolerant can be effectively used to render parallel applications fault tolerant.

Of the various paradigms developed for fault-tolerant distributed computing, passive replication and checkpointing are probably the most appropriate for parallel computing. In fact, we have seen that passive replication can be viewed as a special case of checkpointing. Modern distributed programming toolkits include the necessary technologies for implementing a wide spectrum of fault tolerance techniques.

While technologies such as ISIS are sufficient for fault-tolerant parallel computing, they still require extensive distributed computing expertise to program with. Higher-level interfaces are required if fault tolerance is to be used widely in parallel applications. Paralex represents one such interface. By carefully selecting the programming constructs, fault tolerance can be added automatically to parallel applications. Paralex is proof that this can be accomplished without unreasonable penalties in performance.

Acknowledgements My thinking and appreciation of the material presented in this paper have benefited from discussions with my colleagues Ken Birman, Keith Marzullo, Fred Schnieider and Sam Toueg over the last several years.

References

[1] W. B. Ackerman. Data Flow Languages. *IEEE Computer,* February 1982, pp. 15–22.

[2] Ö. Babaoğlu. On the Reliability of Consensus-Based Fault-Tolerant Distributed Computing Systems. *ACM Trans. on Computer Systems*, vol. 5, no. 3, November 1987, pp. 394–416.

[3] Ö. Babaoğlu and R. Drummond. Streets of Byzantium: Network Architectures for Fast Reliable Broadcasts. *IEEE Trans. on Software Engineering*, vol. SE-11 no. 6, July 1985, pp. 546–554.

[4] Ö. Babaoğlu, L. Alvisi, A. Amoroso and R. Davoli. Mapping Parallel Computations onto Distributed Systems in Paralex. In *Proc. IEEE CompEuro '91*, Bologna, Italy, May 1991.

[5] Ö. Babaoğlu, L. Alvisi, S. Amoroso and R. Davoli. Paralex: An Environment for Parallel Programming Distributed Systems. Technical Report UB-LCS-91-01, Laboratory for Computer Science, University of Bologna, Bologna, Italy, April 1991.

[6] R. G. Babb II. Parallel Processing with Large-Grain Data Flow Techniques. *IEEE Computer,* July 1984, pp. 55–61.

[7] J. F. Bartlett. A NonStop Kernel. *Proc. Eighth Symposium on Operating Systems Principles*, Asilomar, California, December 1981, pp. 22–29.

[8] A. Beguelin, J. J. Dongarra , G. A. Geist, R. Manchek and V. S. Sunderam. Graphical Development Tools for Network-Based Concurrent Supercomputing. In *Proc. Supercomputing '91*, Albuquerque, New Mexico, November 1991.

[9] K. P. Birman. Replication and Fault-Tolerance in the ISIS System. In *Proc. Tenth Symposium on Operating System Principles*, Orcas Island, Washington, December 1985, pp. 79–86.

[10] K. P. Birman and T. A. Joseph. Exploiting Virtual Synchrony in Distributed Systems. *Proc. Eleventh Symposium on Operating System Principles*, Austin, Texas, November 1987.

[11] K. Birman, R. Cooper, T. Joseph, K. Kane and F. Schmuck. The ISIS System Manual. Department of Computer Science, Cornell University, Ithaca, New York.

[12] A. D. Birrell and B. J. Nelson. Implementing Remote Procedure Calls. *ACM Trans. on Computer Systems*, vol. 2, no. 1, February 1984, pp. 39–59.

[13] A. Borg, W. Blau, W. Graetsch, F. Hermann and W. Oberle. Fault Tolerance Under UNIX. *ACM Trans. on Computer Systems*, vol. 7, no. 1, February 1989, pp. 1–24.

[14] K. M. Chandy and L. Lamport. Distributed Snapshots: Determining Global States in a Distributed System. *ACM Trans. on Computer Systems*, vol. 3, no. 1, February 1985, pp. 63–75.

[15] J. M. Chang and N. F. Maxemchuck. Reliable broadcast protocols. *ACM Trans. Computing Systems*, vol. 2, no. 3, August 1984, pp. 251–273.

[16] F. Cristian. Understanding Fault-Tolerant Distributed Systems. *Comm. of the ACM*, vol. 34, no. 2, February 1991, pp. 57–78.

[17] F. Cristian, H. Aghili and R. Strong. Atomic Broadcasts: From Simple Message Diffusion to Byzantine Agreement. In *Proc. 15th International Symposium on Fault-Tolerant Computing*, July 1985, pp. 200–206.

[18] W. J. Dally, A. Chien, S. Fiske, W. Horwat, J. Keen, M. Larivee, R. Lethin, P. Nuth, S. Wills, P. Carrick and G. Fyler. The J-Machine: A Fine-Grain Concurrent Computer. In *Proc. 1989 IFIP Congress*, North-Holland, August 1989, pp. 1147–1153.

[19] J. Gray, P. McJones, M. Blasgen, B. Linsay, R. Lorie, T. Price, F. Putzolu, and I. Traiger. The Recovery Manager of the System R Database Manager. *ACM Computing Surveys*, 13(2), June 1981, pp. 223–242.

[20] D. W. Hillis. *The Connection Machine.* MIT Press, Cambridge, Massachusetts, 1985.

[21] IEEE Special Report. Gigabit Network Testbeds. *IEEE Computer,* vol. 23, no. 9, September 1990, pp. 77–80.

[22] D. B. Johnson and W. Zwaenepoel. Sender-Based Message Logging. In *Proc. 17th International Symposium on Fault-Tolerant Computing*, Pittsburgh, Pennsylvania, July 1987, pp. 14–19.

[23] R. Koo and S. Toueg. Checkpointing and Rollback-Recovery for Distributed Systems. *IEEE Trans. on Software Engineering*, vol. SE-13, no. 1, January 1987, pp. 23–31.

[24] L. Lamport. Time, clocks and the ordering of events in a distributed system. *Comm. of the ACM,* vol. 21, no. 7, July 1978, pp. 558–565.

[25] L. Lamport, R. Shostak and M. Pease. The Byzantine Generals Problem. *ACM Trans. Programming Languages and Systems*, vol. 4, no. 3, July 1982, pp. 382-401.

[26] B. Lampson and H. Sturgis. Atomic Transactions. In Distributed Systems: An Advanced Course. *Lecture Notes in Computer Science*, vol. 105, Springer Verlag, 1981.

[27] B. Liskov, M. Day, M. Herlihy, P. Johnson, G. Leavens, R. Scheifler and W. Weihl. Argus Reference Manual. Technical Report MIT/LCS/TR-400, Massachusetts Institute of Technology, Cambridge, Massachusetts, November 1987.

[28] S. Mishra, L. L. Peterson and R. D. Schlichting. A Membership Protocol Based on Partial Order. In *Proc. 2nd IFIP International Working Conference on Dependable Computing for Critical Applications*, Tucson, Arizona, February 1991.

[29] V. Nicola and A. Goyal. Limits of parallelism in fault-tolerant multiprocessors. In *Proc. 2nd IFIP International Working Conference on Dependable Computing for Critical Applications*, Tucson, Arizona, February 1991, pp. 65–72.

[30] Schneider, F.B. Abstractions for fault tolerance in distributed systems. *Information Processing 86*, H.-J. Kugler (ed.) Elsevier Science Publishers B.V. (North Holland), pp. 727–733.

[31] Schneider, F.B. The state machine approach: A Tutorial. *ACM Computing Surveys*, vol. 22, no. 4, December 1990, pp. 299-319.

[32] S. K. Shrivastava, G. N. Dixon, G. D. Parrington, F. Hedayati, S. M. Wheater and M. C. Little. The Design and Implementation of Arjuna. Technical Report, Computing Laboratory, University of Newcastle upon Tyne, March 1989.

[33] A. Z. Spector, D. S. Daniels, D. J. Duchamp, J. L. Eppinger and R. Pausch. Distributed Transactions for Reliable Systems. In *Proc. Tenth Symposium on Operating System Principles*, Orcas Island, Washington, December 1985, pp. 127–146.

[34] R. E. Strom and S. Yemini. Optimistic Recovery in Distributed Systems. *ACM Trans. on Computer Systems*, vol. 3, no. 3, August 1985, pp. 204–226.

[35] D. Taylor and G. Wilson. The Stratus System Architecture. In *Dependability of Resilient Computers*, T. Anderson (Ed.), Blackwell Scientific Publications, Oxford, England, 1989, pp. 222–236.

[36] Y. C. Tay. The Reliability of (k,n)-Resilient Distributed Systems. In *Proc. of the Fourth Symposium on Reliability in Distributed Software and Database Systems*, Silver Spring, Maryland, October 1984, pp. 119–122.