# Support for Reliable Distributed Computing

Gail E. Kaiser
Wenwey Hseush
Columbia University
Department of Computer Science
New York, NY 10027

June 1988

CUCS-355-88

## Abstract

This technical report consists of two papers describing support for reliable distributed computing. *Data Path Debugging: Data-Oriented Debugging for a Concurrent Programming Language* explains our goal of data-oriented debugging, and then presents *Data Path Expression* (DPE) debugging, our approach to data-oriented debugging for concurrent programming languages. DPE is being implemented as part of MD, the MELD Debugger. *A Network Architecture for Reliable Distributed Computing* proposes a *reliable distributed environment* (RDE) based on an efficient and reliable extension to datagram communications. It gives simulation results for coupled relations based on different algorithms, node failure rate, recovery rate, message sending rate and data missing rate, and to illustrate the behavior of distributed systems constructed using our view section model on top of RDE.

Prof. Kaiser is supported in part by grants from AT&T, IBM, Siemens and Sun, in part by the Center of Advanced Technology and by the Center for Telecommunications Research, and in part by a DEC Faculty Award. Mr. Hseush is supported in part by the Center of Advanced Technology.

# Data Path Debugging: Data-Oriented Debugging for a Concurrent Programming Language

Wenwey Hseush  Gail E. Kaiser

Columbia University
Department of Computer Science
New York, NY 10027

## 1. Introduction

One trend in new programming languages, whether sequential or concurrent, is to include facilities that permit problem solving to be directed from the viewpoint of the problem domain. Object-oriented and dataflow languages are two prominent examples. A related trend in symbolic, debuggers is for the debugger's command language to be both conceptually and syntactically close to the target programming language. These two trends are combined in data-oriented debugging, a form of problem-directed debugging.

We first describe our goal of data-oriented debugging, and then present *Data Path Expression* (DPE) debugging, our approach to data-oriented debugging for concurrent programming languages. DPE debugging is an extension of Bruegge's generalized path expression debugging [6, 7]. In DPE debugging, (1) the debugger is aware of data flow as well as control flow and/or message flow, (2) the debugging language can express breakpoints, single stepping, traces and so on in terms of the data as well as the control, and (3) the debugger can automatically validate both sequential and concurrent paths.

## 2. Data-Oriented Debugging

Data-oriented debugging is an approach to problem-directed debugging since (1) data in programs usually represent abstract entities in the problem domain, (2) data is usually more sensible and meaningful than control to program users, and (3) bugs are usually incarnated in inappropriate data values. Traditional debugging is control-oriented, so in order to compare actual with expected behavior, the user must first interpret the abstract entities of the problem in terms of control structures, which are bound to the syntax and semantics of the particular programming language. The user develops, in his or her mind, an intermediate form between the problem (domain) and the program.

We believe that such intermediate forms often retard users from finding bugs directly and quickly. For example, in cases such as logical errors, the intermediate form in a user's mind may be as incorrect and inappropriate to represent the problem behaviors as the program, since the user of the debugger is usually the same person who wrote the program. Data-oriented debugging eliminates the intermediate forms and provides an almost face-to-face contact between the problem and the program. Problem behaviors can be described to the debugger in terms of data that are meaningful to humans and program behaviors can be reported by the debugger in terms of data that are sensible to humans. A user can repeatedly describe to the debugger the incorrect/correct problem behaviors based on the previous program behaviors reported by the debugger, until the bugs are found. Data-oriented debugging provides a "closer to the problem domain" viewpoint for debugging. We now describe a general form of debugging facilities that are oriented towards the data, but are not specific to data path expressions.

## 2.1. Behavior Data, Histories and Data Events

The first question for debugging is how programmers know something is wrong with their programs. They must sense something unexpected or unusual with the final or intermediate output data, often displayed on the screen. Most misbehaviors are obviously data-related, except perhaps those where an expected event never happens. Let *behavior data* refer to the entities that represent the program behaviors, no matter whether they are expected or not. Expected behavior data can be the objects (or variables) defined in the program, messages, sounds (*e.g.*, beep), graphics displays, screen control (*e.g.*, moving the cursor up and down) and any kinds of intentional signals meaningful to humans, which usually represent the abstract entities of the original problem. Unexpected behavior data can be timeout, unexpected printout, cursor disappearance, robot hands out of control and messages from the operating system (*e.g.*, segmentation fault, I/O error or bus error). Some types of behavior data like screen display are sensible to humans without the help of debuggers; other types of behavior data like variables during execution cannot be understood by humans without debuggers. We concentrate on the values of variables.

The *history* of a given variable, which might represent an abstract entity in the problem domain, is a sequence of the states of the variable in each round of program execution (we refer to each re-execution of a program during testing and debugging as a "round"). For example, the history of the `subtotal` datum of a payroll program might be {20, 30, 44, 70, 95, ... , 155}. The history of a particular philosopher in the dining philosophers problem might be {[NO_FORK, NOT_EAT, NO_FORK], [FORK, NOT_EAT, NO_FORK], [FORK, NOT_EAT, FORK], [FORK, EAT, FORK], ... }.

The entries in such histories can be described by *data events*, denoted by [ *condition* ]. Any two consecutive data events in a history are ordered by time and are different in value. The time intervals between any two consecutive data events are not necessarily the same. The time associated with each data event is the time that the *condition* becomes satisfied. For example, the event that `subtotal` becomes equal to 30 can be described as [subtotal = 30] or [subtotal > 25 and subtotal < 35]. A *standardized* data event has the format [ *behavior_datum* = *state* ] (*i.e.*, [subtotal = 30]). Each program execution can be viewed as a *multiple-history graph* where the histories of global variables are maintained from the beginning to the end of execution, and the histories of dynamic local variables are created at certain times and destroyed at later times. A *standardized* multiple-history graph can be uniquely constructed by using standardized data events for each round of program execution. There is a mapping from any given history to the control flow of the program that generated that history during execution. It is often feasible but usually difficult to deduce the prior control flow from any given data event. The major job of a data-oriented debugger is to keep track of the mappings between the control events and the histories of given variables.

## 2.2. Data-Oriented Debugging for Sequential Languages

Using data-oriented debugging, where expected/unexpected behavior data can be described as data events, certain kinds of bugs are easy to locate. For example, say a programmer senses the existence of bugs in the program by observing the behavior datum `salary`, which is printed as -1000 instead of the expected 3000. He/she starts looking for the actual program error by telling the debugger "As soon as `salary` equals -1000, break and print all the entities that caused `salary` to be -1000". On the next round, the debugger might give more information about behavior data like "The related entities are `subtotal` equal to 30 and `total` equal to -1030". The programmer can first check whether the statement that adds `subtotal` to `total` is correct or not. If so, then <u>one</u> bug has been found. If not, the programmer might eliminate the possibility of incorrect `subtotal` and treat only `total` as the

suspicious behavior datum, and then continue to locate bugs by following the history of `total`. Some other examples are "when the cursor disappears, break", "when a beep occurs, break" or "when the memory at address 0xFFFF is overwritten, break". Some behavior data are difficult to keep track of during execution.

In general, the programmer repeats the following procedure until the bugs are found.

1. **Programmer**: Execute the program and observe suspicious behavior data (*e.g.*, salary).
2. **Programmer**: Describe the suspicious behavior data to the debugger (*e.g.*, "As soon as salary is equal to -1000, break and print all the entities that cause salary to be -1000.").
3. **Programmer**: Re-execute the program with the debugger by following the history of the suspicious behavior data (*e.g.*, the history of salary).
4. **Debugger**: Suspend the program and report (*e.g.*, "The related entities are subtotal equal to 30 and total equal to -1030.").
5. **Programmer**: Check the statements around the break point; sometimes the bug is obvious.
6. **Programmer**: If no bug is found, check all the behavior data that directly causes the suspicious behavior data, eliminate some of them and treat the others as suspicious behavior data. Go to 2.

## 2.3. Data-Oriented Debugging for Concurrent Languages

For debugging sequential programs, one important assumption is deterministic behavior. That is, a program can be re-executed over and over again until bugs are found. In concurrent programming, this assumption frequently does not hold. Given this problem, the first question is what role the debugger can and should play for concurrent programming languages. Some suggestions are:

- Operate the debugger like a video tape recorder. The entire history of program execution can be recorded and then played back (forwards or backwards) as many times as desired without actually re-executing the program. Stu Feldman has described how he does this in hardware (private communication). However, it is unclear how to analyze the vast amount of information once it has been recorded.
- Treat the debugger as a program verification system. This is currently infeasible for small sequential programs, let alone large concurrent ones.
- Require the debugger to find all feasible paths [9], including all possible interleavings, in the course of applying program-based testing techniques [12].
- Use the debugger to force an execution flow identical to the previous one.
- Ask the debugger to detect when the current execution flow is or is not the same as the previous one or an anticipated one.

For practicality, we consider only the latter. For each round of execution, we save a special form of multiple-history graph that represents the program behavior in terms of histories of behavior data. For program execution, the current graph is compared with the graph from a previous execution or with an anticipated flow provided by the user. We discuss this graph further in section 5.

# 3. Path Expression Debugging

Bruegge applied three versions of path expressions to debugging.

**Path expressions** [11]: A regular expression with the operators repetition (*), sequencing (;) and exclusive selection (|). The operands — called path functions — are the names of the operations

defined for a data type. A path function might be a procedure call or a function call. For example,

**Path (Open; (Read | Write)*; Close) End**

states that a file has to be opened first, before an arbitrary sequence of reads and writes is performed, and then closed.

**Predicate path expressions** [1]: The original path expressions augmented with predicates. For example,

**(Consume[TERM(Produce) - ACT(Consume) > 0] | Produce[ACT(Produce) - TERM(Consume) < N])***

states the operational relationship between Consume and Produce. ACT and TERM are history variables, which describe how many times that caller has started to perform the operation (*i.e.*, Produce or Consume) and how many times that caller has terminated the operation, respectively.

**Generalized path expressions**: Predicate path expressions are extended by i) allowing an arbitrary block of statements, including a single statement, to be a path function; ii) allowing the arguments in a predicate to be any identifiers declared in the source program; iii) providing a predefined path function, ***_Assignment***, to refer to a change in the state of a variable (this is the simplest form of data event).

All three versions are control-oriented, where activities in a specified control flow are interpreted as checking points in order to compare with the actual program behaviors.

DPEs extend generalized path expressions, mainly by allowing data events to appear as path functions. DPEs describe the histories of some given behavior data as well as the control flow. DPEs can describe concurrent program behaviors with a powerful set of operators for specifying the concurrent characteristics among data events. A debugger based on DPEs can automatically check the correctness of the multiple histories of a given system, either concurrent or sequential, and violations of the DPEs during program execution cause breaks or other prescribed actions.

Filtering is an important feature for path expressions in debugging [4]. Since data in programs can represent problem entities at different granularities, different DPEs may describe different abstraction levels with respect to the same data and typically mention only a subset of the data in the program. For example, a user may be interested only in the behavior datum `total` but not in `subtotal` at a particular moment. Between two data events related to `total`, some irrelevant events related to `subtotal` might happen. Since we specify the DPE without mentioning the irrelevant behavior data, all irrelevant events are automatically ignored. The irrelevant events will not affect whether or not the specified DPEs match the run-time paths. We give a detailed description of DPEs in the next two sections, focusing on sequential and concurrent aspects, respectively.

## 4. Sequential Data Path Expressions

The concept of sequential DPEs consists of application of DPEs without consideration of inter-relationships among concurrent units. A sequential DPE is a regular expression, where operands are data or control events, either of which can be annotated with immediate actions. An event in DPEs may or may not have a predicate attached.

The operators in sequential DPEs are repetition (*), sequencing (;) and selection/or (|). Since in sequential computing, only one event happens at a time, the selection operator means exclusive selection (exclusive or). In concurrent DPEs, we distinguish between inclusive and exclusive or.

A new operator, permutation (,), is also considered as an operator in DPEs, even though it is not an operator in regular expressions. The permutation operator specifies that the events associated with it may happen in any order. For example, **(READ,WRITE)** is equivalent to **(READ;WRITE)|(WRITE;READ)**. This brings the users some convenience in specifying DPEs where the order of a set of events is known to

be irrelevant to the bugs. The permutation operator is more significant for concurrent DPEs, as will be discussed shortly.

There are three categories of events in DPEs:

Data events, denoted [ *condition* ], are the events that the *condition* becomes true. Any identifier declared in the source program can be used inside []; additional identifiers representing debug attributes (explained later) can also appear. A standardized data event has the format [ *behavior_datum* = *state* ]. For example, [flag = 1] is the event that flag becomes equal to one; [flag > 0] describes the same event. In the case of sequential computing, [X > 0 and Y > 0] is the event that either i) X is larger than 0 and then Y becomes larger than 0, or ii) Y is larger than 0 and then X becomes larger than 0.

Control events, denoted by identifiers that might be function names, procedure names or attached to blocks of statements, represent activities in control flow. For example, READ is the event that the function named READ is called.

Message events, denoted by ? and ! followed by message identifiers, are the events of the sending and the receiving of the messages. !*message* refers to the event of sending the message and ?*message* refers to the event of receiving the message.

The semantics of control events and message events differ from one language to another, and in some programming languages there is some level of overlap between control events and message events. For example, in some object-oriented languages, message events subsume function calls and procedure calls. However, the different semantics of these events do not affect the results in this paper.

Two notations on behavior data are used to describe some special cases for data events: (1) A behavior datum without any relational operator (*i.e.*, =, >, <, <=, >=, !=) denotes an access to the behavior datum. For example, if X is a variable in a program, [X] is the event of an access to X. [X or Y > 0] is the event of either an access to X or that Y becomes larger than 0. The event [X and Y] never occurs in sequential computing. (2) The notation of (') postfixing a behavior datum denotes the previous value of the behavior datum. For example, `counter` is a behavior datum and `counter'` is the previous value of `counter`; the program behavior of positively incrementing `counter` by one can be specified as [counter = counter' + 1].

A *conditional event* is any kind of event (data event, control event or message event) followed by a predicate. The format of predicates is [ *condition* ], the same as for data events. Conditional events are recognized when the event in the first part happens in a context where the condition in the predicate is satisfied. For example, if READ is a control event, then READ[flag = 0] is the event that READ is invoked while `flag` is equal to 0. [X > 0][Y > 0] is the event that X becomes larger than 0 under the condition that Y is already larger than 0. Note that this is different from [X > 0 and Y > 0], since the former does not describe the case that X becomes larger than 0 before Y is larger than 0. Predicates play an important role in event recognition.

The format of an *immediate action* is { *statements* }. Immediate actions in DPEs are analogous to semantic routines in syntax-directed translations (*e.g.*, YACC [14]). An immediate action will be evaluated as soon as the corresponding path is recognized, in the same manner that a semantic routine is invoked as soon as the non-terminal is recognized in syntax-directed translation. For example, [X > X']{ counter := counter + 1; } will increment counter by one when the event, [X > X'], is recognized. For more obvious debugging purposes, [X != X']{ print(X) } will print out the value of X whenever X is changed. [QSIZE > N]; ?MESSAGE { break; } will suspend the execution whenever the message is received after `QSIZE` is larger than N.

Immediate actions play two roles in DPEs. First, the debugger is able to change the program's behavior

automatically, thus it changes paths at run-time. Sometimes programmers want to force the program to behave in a particular way for debugging purposes. With proper use of immediate actions (*e.g.*, if-then-else statements can be used in immediate actions), a program execution can be forced in some direction, typically where the programmer thinks bugs are most likely to be found. In traditional debuggers, the programmer would have to break the program execution, assign values to some variables, then resume execution; in DPE debugging, value-assignments are automatically done as soon as some patterns of path are matched. The decision of selecting a desired path also can be made interactively by setting the I/O interface in the immediate actions. One example is to check a 'context-free' path by using a DPE such as **[X != X']{ push(X);}*; [Y != Y']{ Z := pop(); IF (Y != Z) THEN break; }***, which (i) saves the value of X on a stack (*i.e.*, push(X)) whenever X is changed, (ii) retrieves X's value into Z from the stack (*i.e.*, Z := pop()) when Y is changed, (iii) checks whether Y and Z are equal or not, (iv) if not then breaks the program execution. One legal data path is [X=1] [X=2] [X=3] [Y=3] [Y=2] [Y=1].

Second, additional debugging control can be achieved. For example, we like to know how many times that behavior datum X is accessed before the condition [flag = 1] is satisfied. The DPE is **{ counter := 0; };[X]{ counter := counter + 1; }*;[flag = 1]{ print counter; }**. Here, counter is not a behavior datum, but a *debug attribute* that is attached to this particular DPE. The difference between behavior data and debug attributes is that debug attributes are not declared in the source program to represent some abstraction of program behavior, but instead 'declared' to the debugger. In this example the first action is attached to a null event, so it will be evaluated as soon as the round begins.

A set of DPEs can be specified in order to describe different patterns of histories among behavior data. If all events happen within one concurrent unit, from the global view there is only one sequence defining the occurrence of all the events (*i.e.*, standardized multiple-history graphs). From any particular local view described by a single DPE, only part of the events are visible in the sequence, therefore different histories are described according to the interesting program behaviors. Multiple DPEs can describe a set of different histories from different local views. We now discuss the case that events happen in two or more concurrent units.

## 5. Concurrent Data Path Expressions

A concurrent DPE specifies multiple streams of data histories as well as control and message flows among two or more concurrent units. The units may be related to each other through message passing, shared memory and/or data dependency. Some programming languages support interprocess communication and synchronization through message passing, some through shared memory, while others provide both. The data dependency facility is suitable only for those languages that support dataflow, lazy evaluation and/or other features where control flow is based on dependencies among data values — note that this is different from the 'data dependencies' implied by shared memory.

The effects of message passing, shared memory and/or data dependency can be represented (conceptually) in a *partial ordering graph* [17] for each round of execution. A partial ordering graph is a special form of multiple-history graph.

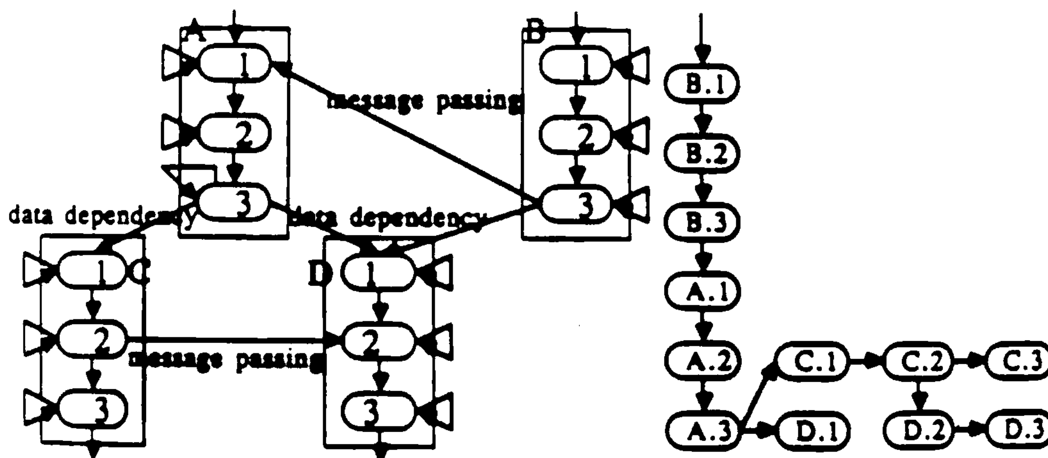
In partial ordering graphs, vertices are events — data events, control events, message events and/or *macro-events* made up of several sub-events. Edges between vertices are directed and represent the relation *affect immediately*: an edge from vertex a to b means that a affects b immediately. Even though an event cannot affect itself, the reflexive property is explicitly defined in the relation. That is, for any vertex a, an edge is directed from a to a; we explain why this is necessary later on. A *standardized* partial ordering graph is a partial ordering graph constructed without using any non-standardized data

events.

A standardized partial ordering graph can be constructed by the following procedure:

1. (**inherent parallelism**) Assume a set of concurrent units, let each unit be an event (a vertex) that may contain a sequence of sub-events; therefore, all the events are potentially executed in parallel.
2. (**data dependency**) In most programming languages, in which the execution ordering is not determined by data dependency, this step can be skipped. The execution ordering between event a and b is determined in the way that if the output data of event a affects the input data of event b, then a occurs before b; thus an edge is directed from a to b. The events without an incoming edge will be executed as soon as the program begins. If there is no data dependency between them, a and b are assumed to happen in parallel. Circularities in graphs are considered as logical errors; this is not an issue in this paper.
3. (**inherent sequencing**) Zoom all events so that i) a sequence of sub-events substitute for an event, for all events; ii) the edges into an event are connected to the first sub-event; iii) the edges out of an event are connected to the last sub-event.
4. (**message passing**) If sub-event a is the sending of a message and b is the receipt of the same message, then an edge is directed from a to b.
5. (**shared memory**) If sub-events W1 and W2 are two consecutive events that modify the same portion of shared memory, for any sub-event R that sees the value(s) of the portion of shared memory that W1 changed, there is an edge directed from W1 to R and an edge from R to W2. An edge from W1 to W2 is necessary if there is no sub-event R between W1 and W2.
6. (**reflexive property**) For each sub-event a, there is an edge from a to a, even though the event cannot affect itself.

Figure 1 illustrates an example with four concurrent units, A, B, C and D. Data dependency requires A and B to execute before C and D. C will not be executed until A completes and D will not be executed until both A and B complete. Each event has three sub-events, 1, 2 and 3. The sub-event B.3 sends a message to A.1 and C.2 sends a message to D.2.



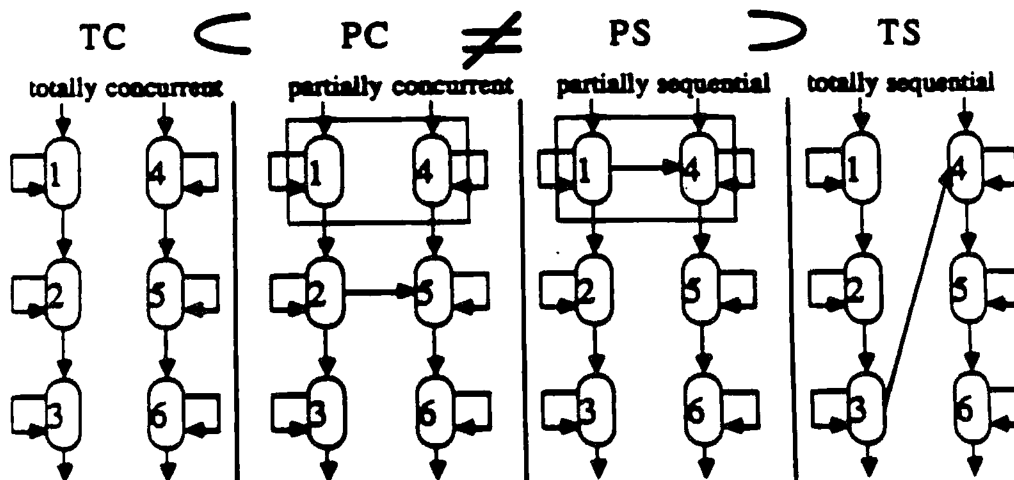

1. Partial ordering graph and execution order

Depending on the granularity of the events that the user is interested in, different partial ordering graphs are constructed. Non-determinism in parallel/distributed systems results in different partial ordering graphs in different rounds of execution for the same program. As a starting point for discussing concurrent DPEs, we assume there exists (conceptually) a standardized partial ordering graph for each

round of execution and the graph is non-circular.

Two events are *partially concurrent*, if and only if there is no path between them in the partial ordering graph. Two distinct sets of events are partially concurrent, if and only if there exists two events, which are from different sets, that are partially concurrent. These two distinct sets of events may contain common events, but one set cannot contain the other. Two distinct sets of events are *totally concurrent*, if and only if all possible pairs of events, one event from one set and the other event from the other set, are partially concurrent. Two events are *partially sequential*, if and only if there exists a path between them in the partial ordering graph. Two distinct sets of events are partially sequential, if and only if there exists two events, which are from different sets, that are partially sequential. Two distinct sets of events are *totally sequential*, if and only if all possible pairs of events, one event from one set and the other event from the other set, are partially sequential and all paths between two sets have the same direction. Partial concurrency is not equivalent to partial sequentiality, even though they often describe the same cases.

Figure 2 depicts four sets of partial ordering graphs — TC, PC, PS and TS. The graphs are defined on two sets of events, A = {1, 2, 3} and B = {4, 5, 6}, which are related with total concurrency, partial concurrency, partial sequentiality and total sequentiality, respectively. The execution orders are 1 → 2 → 3 in A, and 4 → 5 → 6 in B. TC contains only one graph with no path between A and B. PC contains all the graphs that at least one pair of events in AxB have no path between them. (TC ⊆ PC) PS contains all the graphs where at least one pair of events in AxB have paths between them. TS contains only two graphs, with an edge from 3 to 4 and an edge from 6 to 1. (TS ⊆ PS) Some other properties are (i) PC ≠ PS, (ii) PC ∪ TS includes all possible partial ordering graphs, (iii) PC ∩ TS = ∅.




2. Concurrency relationships

Concurrent DPEs specify a superset of what sequential DPEs can specify. Concurrent DPEs have the same definitions of events, predicates and immediate actions as sequential DPEs. The major difference is the definitions of the operators, which may be applied to macro-events as well as to data, control and message events. Four new operators are introduced in concurrent DPEs:

- **exclusive or (+)** means exactly one associated event should occur.
- **partial concurrency/and (&)** means two associated events are partially concurrent.
- **total concurrency (&&)** means two associated events are totally concurrent.
- **partial sequentiality (:)** means two associated events are partially sequential.

Users are usually interested only in partial concurrency and total sequentiality. Two sets of events must

be either partially concurrent or totally sequential, but not both. For example, [X];[Y];[Z] is a set of events and [S];[Y];[T] is another set of events. These two macro-events are partially concurrent, because there is an edge from the event [Y] in the first macro-event to the event [Y] in the second. Another example, [X];[Y];[Z] is a macro-event and [Z];[S];[T] is another. These two are totally sequential, because there is a path from any event in the first to any event in the second macro-event. Without the reflexive property, we cannot say these are totally sequential. This is why we include reflexive edges in the construction of partial ordering graphs.

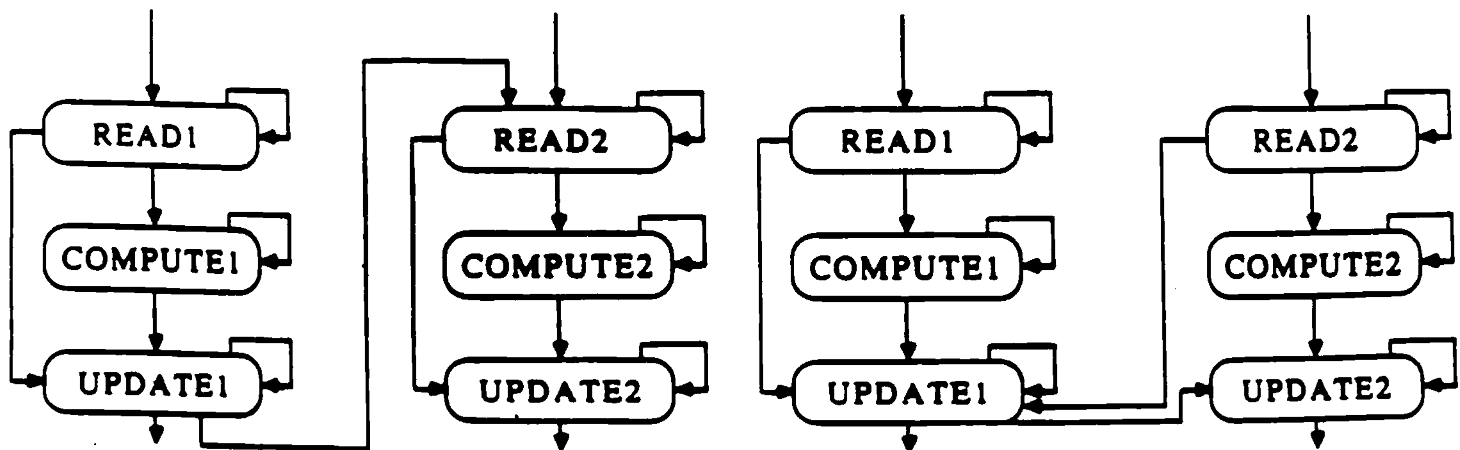
All operators appearing in sequential DPEs have slightly different meanings for concurrent DPEs.

- **sequencing** (;) means two consecutive events are totally sequential.
- **repetition** (*) means two consecutive events are totally sequential.
- **selection/or** (|) does not mean exclusive selection in concurrent computing; instead, it means at least one associated event should occur. If a and b are events, a|b means a+b+(a&b)+(a&&b), which is one of the cases: i) only a happens; ii) only b happens; iii) a and b are partially concurrent; iv) a and b are totally concurrent.
- **permutation** (,) if a and b are events, (a,b) means (a;b)+(b;a)+(a&b)+(a&&b), which is one of the cases: i) a happens before b; ii) b happens before a; iii) a and b are partially concurrent; iv) a and b are totally concurrent. Permutation is important since there may be two or more legal execution orders, each producing the correct results. DPEs were originally formulated without the permutation operator, but we found that timing order problems made it tedious for programmers to specify suitable DPEs for concurrent computing.

For example, assume two concurrent transactions, `T1` and `T2`, applied to a shared database, X. `T1` contains three sequential events, `READ1`, `COMPUTE1` and `UPDATE1`, and `T2` also contains three sequential events, `READ2`, `COMPUTE2` and `UPDATE2`. Because there is a bug in the concurrency control in the database, a programmer wants to detect possible cases where two transactions happen at the same time. He/she can specify a concurrent DPE

**(READ1;COMPUTE1;UPDATE1)&(READ2;COMPUTE2;UPDATE2)(break;)**

For each round of program execution, a partial ordering graph will be built through shared memory and the DPE will be checked according to the graph. If `READ2` happens after `UPDATE1`, then the partial ordering graph constructed at run-time is as in figure 3. These two transactions are totally sequential (*i.e.*, ;). If `READ2` happens between `READ1` and `UPDATE1`, then the partial ordering graph is as in figure 4. If these two sets are partially concurrent (*i.e.*, &), then the debugger breaks the execution.




3. Total sequentiality

4. Partial concurrency

## 6. Related Work

Path expressions were first introduced by Campbell and Habermann as a formalism for specifying process synchronization [8]. Lauer and Janicki used path expressions in COSY [19, 18, 13] to simplify the study of synchronous aspects of concurrent systems. Balraj and Foster use path expressions to specify synchronization requirements as part of the input to Miss Manners [3], a silicon compiler for synchronizers. Balraj and Foster allow flags to be attached to path expressions, where the states of the system can be changed by setting flags; thus, the execution path can be selected at run-time. This can be considered an extreme case of DPEs, if we treat flags as behavior data.

Bates' Event Based Behavioral Abstraction (EBBA) [4, 5] provides a high-level viewpoint similar to ours, with the goal of providing facilities that allow the user to make a meaningful comparison between models of actual and expected system behavior. Bates views a system's behavior as a stream of primitive event occurrences and provides powerful operators for composing primitive events into high-level events that can be recognized by the distributed debugger. Unlike data path expressions, however, EBBA does not take any stand on data *versus* control orientation, but instead requires that all primitive events be explicitly signaled in the target program.

Baiardi's Event Specifications and Behaviour Specifications [2] provide a similar high-level viewpoint for formally specifying program behaviors, but there the notion of events is restricted to a small set of synchronization constructs. Gordon's Timing Graphs [10], which are useful for finding timing errors in distributed systems, are restricted to message sending and receiving events.

## 7. Conclusions

We have extended Bruegge's generalized path expression debugging from control-oriented to data-oriented and from sequential to parallel and distributed programs. Our result, data path expression debugging, is a step towards debugging of concurrent programs from the viewpoint of the problem domain rather than from the artifacts of the particular programming language.

An implementation of data path expression debugging, called MD for MELD Debugger, is currently in progress. MELD [15, 16] is an experimental multiparadigm language combining object-oriented, dataflow and transaction programming. The object-oriented aspect supports large grain (distributed) parallelism with both synchronous and asynchronous message passing among remote objects, while the dataflow aspect supports fine grain (MIMD) parallelism among the statements within a method. The implementation of MELD simulates the dataflow within a process, but supports distribution of objects across a network of Suns and Vaxen running Berkeley Unix. Transaction processing facilities are currently being integrated to provide better control over interactions both among methods applied to the same object and among objects.


## Acknowledgements

This research is supported in part by grants from AT&T Foundation, IBM, and Siemens Research and Technology Laboratories, in part by the New York State Center of Advanced Technology — Computer & Information Systems and by the Columbia University Center for Telecommunications Research, and in part by a Digital Equipment Corporation Faculty Award.

## References


[1] Sten Andler.
*Predicate Path Expressions: A High-Level Synchronization Mechanism.*
PhD thesis, Carnegie Mellon University, August, 1979.
CMU-CS-79-134.

[2] F. Baiardi, N. De Francesco, E. Matteoli, S. Stefanini, G. Vaglini.
Development Of A Debugger For A Concurrent Language.
In *ACM SIGSoft/SIGPlan Software Engineering Symposium on High-Level Debugging*, pages 98-106. Pacific Grove, CA, March, 1983.
Special issue of *Software Engineering Notes*, 8(4), August 1983.

[3] T.S. Balraj and M.J. Foster.
Miss Manners: A Specialized Silicon Compiler for Synchronizers.
In *Proceedings of the Fourth MIT Conference*, pages 3-20. The MIT Press, April, 1986.

[4] Peter Bates.
*EBBA Modelling Tool a.k.a. Event Definition Language.*
Technical Report COINS 87-35, Computer and Information Science Department, University of Massachusetts, April, 1987.

[5] Peter Bates.
Distributed Debugging Tools for Heterogeneous Distributed Systems.
In *ACM SIGPlan/SIGOps Workshop on Parallel and Distributed Debugging*. Madison, WI, May, 1988.
This proceedings.

[6] Bernd Bruegge and Peter Hibbard.
Generalized Path Expressions: A High-Level Debugging Mechanism.
*The Journal of Systems and Software* 2(3):265-276, 1983.

[7] Bernd Bruegge.
*Adaptability and Portability of Symbolic Debuggers.*
PhD thesis, Carnegie Mellon University, 1985.

[8] R. H. Campbell and A. N. Habermann.
The Specification of Process Synchronization by Path Expressions.
In G. Goos and J. Hartmanis (editors), *Lecture Notes in Computer Science*. Volume 16: *Operating Systems*, pages 89-102. Springer-Verlag, Berlin, 1974.

[9] Phyllis G. Frankel and Elaine J. Weyucker.
Data Flow Testing in the Presence of Unexecutable Paths.
In *Workshop on Software Testing*, pages 4-13. IEEE Computer Society, Banff, Canada, July, 1986.

[10] Aaron J. Gordon and Raphael A. Finkel.
TAP: A Tool To Find Timing Errors In Distributed Programs.
In *Workshop on Software Testing*, pages 154-163. IEEE Computer Society, Banff, Canada, July, 1986.

[11] A.N. Habermann.
*Implementation of Regular Path Expressions.*
Technical Report, Carnegie Mellon University, February, 1979.

[12] William E. Howden.
*Software Engineering and Technology: Functional Program Testing & Analysis.*
McGraw-Hill Book Co., New York, 1987.

[13] Ryszard Janicki.
A Method For Developing Concurrent Systems.
In *Lecture Notes in Computer Science.* Number 167: *International Symposium on Programming,* pages 155-166. Springer-Verlag, Berlin, 1982.

[14] S.C. Johnson and M.E. Lesk.
Language Development Tools.
*The Bell System Technical Journal* 57(6):2155-2175, July-August, 1978.

[15] Gail E. Kaiser and David Garlan.
Melding Software Systems from Reusable Building Blocks.
*IEEE Software* :17-24, July, 1987.

[16] Gail E. Kaiser and David Garlan.
MELDing Data Flow and Object-Oriented Programming.
In *Object-Oriented Programming Systems, Languages, and Applications Conference,* pages 254-267. Kissimmee, FL, October, 1987.
Special issue of *SIGPLAN Notices,* 22(12), December 1987.

[17] Leslie Lamport.
Time, Clocks and the Ordering of Events in a Distributed System.
*CACM* 21(7):558-564, July, 1978.

[18] P. E. Lauer and M. W. Shields.
*Formal Theory of the Basic COSY Notation.*
Technical Report 143, Computer Lab. University of Newcastle upon Tyne, 1979.

[19] P. E. Lauer and M. W. Shields.
Formal behavioural specification of concurrent systems without globality assumptions.
In J. Diaz and I.Ramos (editor), *Lecture Notes in Computer Science.* Number 107: *Proceedings of Internation Colloquium on Formalization of Programming Concepts,* pages 115-151. Springer-Verlag, Berlin, 1981.

# Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging

May 5-6, 1988
University of Wisconsin
Madison, Wisconsin 53706

## Co-Chairmen

Barton Miller, University of Wisconsin-Madison
Thomas LeBlanc, University of Rochester

## Program Committee

Domenico Ferrari, University of California at Berkeley
Richard Rashid, Carnegie Mellòn University
John Sopka, Digital Equipment Corporation
Douglas Terry, Xerox Corporation Palo Alto Research Center
Andre van Tilborg, Office of Naval Research
Jack Wileden, University of Massachusetts

# A Network Architecture for Reliable Distributed Computing

Wenwey Hseush
Gail E. Kaiser

Introduction

Message passing in loosely-coupled distributed systems is becoming increasingly complex, due, in part, to the movement towards large scale distributed systems and intelligent distributed applications. Traditional approaches such as the client-server model are no longer appropriate. Therefore, we propose a Reliable Distributed Environment (RDE) based on an efficient and reliable extension to datagram communications. The "coupled relation" is introduced to measure the degree to which distributed environments are reliable. "View sections", a programming construct that protects against changes in node status (available or not), as support for distributed computing tasks, are also presented. In addition, we give simulation results for coupled relations based on different algorithms, node failure rate, recovery rate, message sending rate and data missing rate to illustrate the behavior of distributed systems constructed using our view section model on top of RDE.

A Reliable Distributed Environment (RDE) is a collection of loosely-coupled distributed nodes [1] where the environment ensures reliable communication and close view. Reliable communication implies that messages are safely delivered if the destination nodes are functionally working at the moment of message arrival, thus protecting against link failures. Close view provides a clear picture of the near current environment to protect against node failures (process deaths, machine failures [2], or any temporary functional failures on nodes). The "view" in "close view" means the global view, which is the global status of the environment; the "close" means "near past", since the contents of the current global view are impossible to collect through message passing. Close view implies precise prediction of node status. We use the term "node" to refer to a "process" in the transport layer in order to distinguish from a "host" in the network layer.

It would be unnecessary for us to propose RDE if the complexity of message passing had remained as simple as in most traditionally distributed applications, to which the client-server model [2–5] has been applied successfully. The client-server model implies end-to-end communications between two different nodes, which need not know the status of any nodes except each other's. The notion of close view is becoming important since the complexity of message passing is dramatically increasing in large scale and/or intelligent distributed systems, which both require more complicated communication patterns. The client-server relationship no longer holds, and failure to predict network-wide node status results in severe degradation of performance.

0890-8044/88/0007-0028 $01.00 © 1988 IEEE

We propose a programming framework, the view section model, in which to construct reliable distributed computing tasks on top of RDE (see Figure 1). View sections protect against change of the global view in a manner analogous to how critical sections [6] protect against change of shared memory. A view section defines a period of time and a sequence of instructions during which the global view should remain the same to maintain the correctness of the computation performed by the instructions. The fact is, however, that the global view changes from time to time as nodes fail and are restored, even during view sections. We handle this by invoking an application-specific compensation function via an immediate notification generated by RDE when it senses a change of the global view. The compensation function decides what to do to preserve the view section. Further work is required to construct a full transaction mechanism [7, 8] based on the view section. Note, therefore, that we are not concerned here with the issues of reliable distributed databases.

## *Complicated Communication Patterns*

In most loosely-coupled distributed systems, typical communication patterns like the client-server model and complete connections in small domains are so simple that the existing transport layer protocols (e.g., TCP [9]) can perform efficiently and reliably. In large-scale and intelligent distributed systems, communication patterns are much more complicated than the conventional models can handle. In this article, we use the term "virtual circuit" to refer to an end-to-end communication link in the transport layer, where connection between two ends (ports) is required throughout the whole period of communication in order to ensure

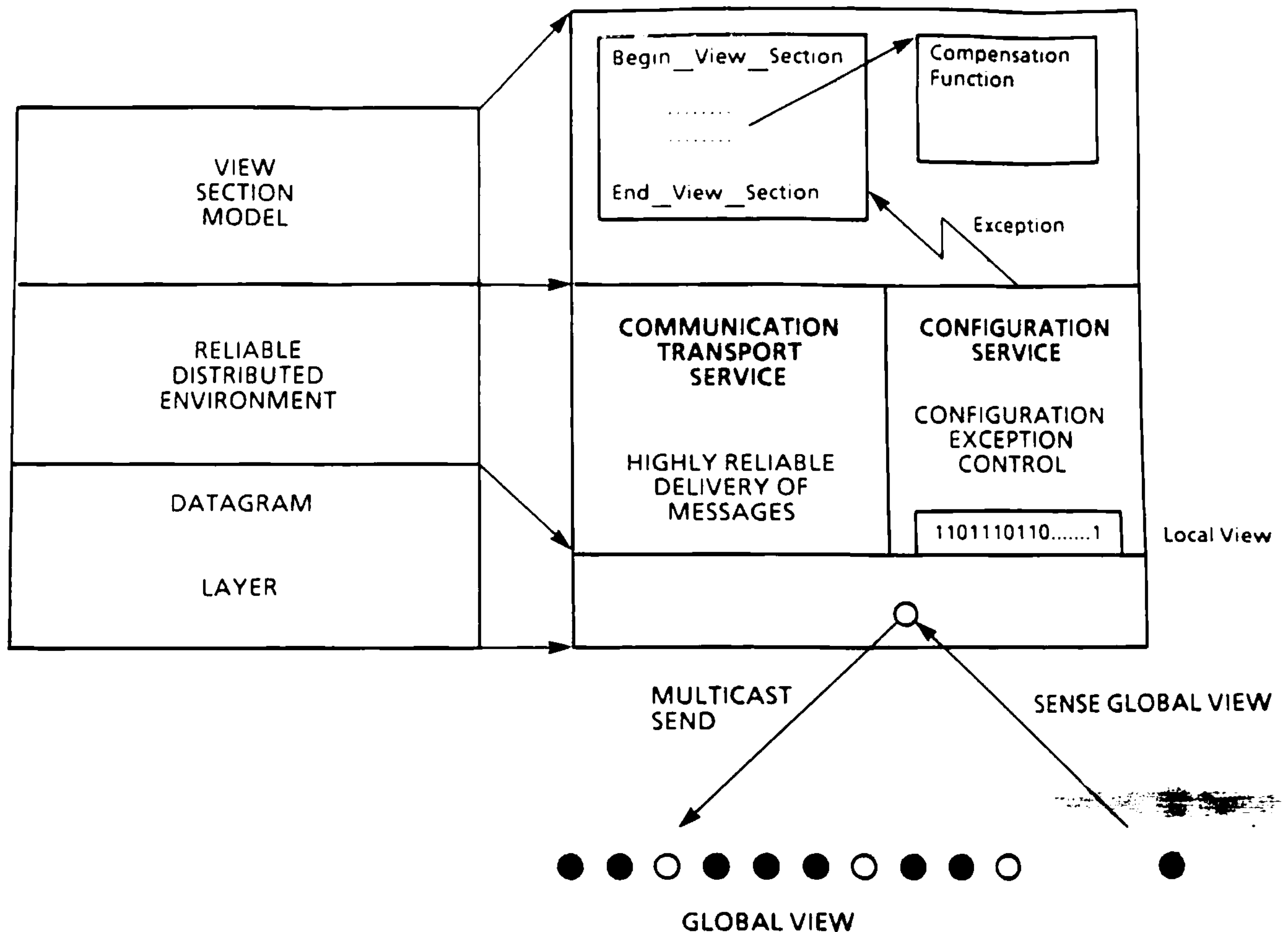



*Fig. 1. A network architecture for reliable distributed computing.*

reliable communication and close view (between two nodes). We use "datagram" to refer to a connectionless communication medium in the transport layer, where a remote address is required for each message send, and neither the status of remote nodes nor whether messages are safely delivered is known to the sender nodes. Some common communication patterns are described below.

## Trivial Communications

Communications randomly and frequently take place among the possible pairs of nodes. That is, i) the number of nodes associated with a communication is small, ii) the number of independent communications is large, and iii) the communication relations among nodes vary from time to time. Typically, instead of using datagram, virtual circuits are used because of the necessity of reliable message delivery. Unfortunately, it is inappropriate to use virtual circuits for end-to-end communications in this case, because the cost of connecting reliable virtual circuits is comparatively higher than that of passing messages. The performance is degraded dramatically.

## Communications in the Large

In some large-scale distributed systems, i) the number of nodes associated with a communication is large and variant, and ii) the configuration of the systems are dynamic [10], where nodes come and go without affecting the rest of a system. Again, it is inappropriate to use virtual circuits in this case, because the complexity of the number of I/O ports, O(N), complicates each node. The complexity of O(N) looks reasonably good but in fact it is not, since I/O resources are limited in most operating systems. For example, in Berkeley Unix™ the number of file descriptors that can be associated with I/O ports (i.e., stream sockets [11, 12]) is quite small. For each file opened as an I/O channel, the kernel reserves a memory buffer for storing incoming data packets and pays attention (CPU time) to detecting the failures of connections and nodes. These costs are expensive.

Unix™ is a trademark of AT&T.

### Multicast Transport Services

In some distributed systems, nodes communicate with each other through multicasting. Grouping, reliable message delivery and exception control are important. Unfortunately, multicast transport services are not supported in most existing transport layer network environments. Even when supported in a network environment, unreliability is usually a problem.

There is a distinction between what virtual circuits and datagrams can provide for applications; virtual circuit supports those applications that need reliable one-to-one communications, while datagram supports those applications that need many-to-many communications without worrying too much about reliability. Neither of them can ensure reliable many-to-many communications. RDE fills this need and solves the problem of complicated communication patterns.

#### *Datagram Layer*

RDE appears to be a third type of transport service (the other two are virtual circuit and datagram). Actually, RDE is built on top of a datagram layer (see Figure 1). Datagram eliminates the problems of high I/O ports complexity and expensive virtual circuit connections, so it seems to be a better approach for large-scale, loosely-coupled distributed systems. The disadvantages and advantages of using datagram have to be pointed out to explain our design of RDE using datagram. For most existing datagram transport services (e.g., User Datagram Protocol), unreliability is the major problem. Datagram packets may be delivered multiple times or out of sequence, or not delivered at all. A sender neither knows the status of the destination nodes, nor can be assured that the message packets have been safely received. A positive acknowledgment scheme is often used to ensure safe delivery of messages. The most pleasant aspect of datagram is that only one I/O port is needed for each node to send and receive messages, which is particularly important when I/O resources are limited. Therefore, it makes more sense to implement a protocol on top of datagram to ensure reliable delivery of messages rather than implementing a protocol on top of virtual circuit to reduce the number of ports used.

The simplicity of one I/O port is a very convincing reason for large-scale distributed systems to use datagram. This results in the trend for intelligent distributed applications to handle increasingly complex patterns of message communications using one I/O port for each node, which listens to or talks to all other nodes without building end-to-end connections, and expecting a small degree of unreliability.

## Reliable Distributed Environment

The goal of RDE is to extend the reliability between a pair of nodes, which has been promised by virtual circuit, to reliability among a group of nodes. As mentioned above, RDE is defined as a collection of loosely-coupled distributed nodes that ensures reliable communication and close view. Each distributed node has a static view, called the local view, to reflect the status of the environment. In RDE, a local view is a bit string—called configuration bits—that indicates the status of each node (active or inactive), in a designated order according to the nodes domain of the distributed environment. The global view is imagined configuration bits constructed from the status of all nodes in the same order as the local view.

Reliable communication can be achieved by using a $k$-retry/time-out and positive acknowledgment scheme, which ensure messages are safely delivered at destination nodes. "Delivered at destination nodes" should be distinguished from "received by destination hosts" and "received by destination nodes". "Received by destinations hosts" does not imply the existence of the destination nodes, and "received by destination nodes" implies that messages are explicitly read by the application programs, which might take arbitrarily long and cause the sender nodes to time out.

We can think of "host" as a local post office and "node" as a house with a mailbox; then mail (message) is delivered to the mailbox of the destination house and the postman sends back an acknowledgment. Close view in RDE means precise prediction of node status. That is, RDE approximates local views to the global view. Due to the nature of message passing, local views can reflect, at best, the environment status in the near past. This can be achieved by running predicting algorithms in RDE, which will be described in the "Distributed Predicting Algorithms" section.

Reliable communications and close view are not mutually independent. Reliable communications, together with the effect of close view, ensure reliable delivery of messages, which requires reliable links and healthy nodes (functionally working). That is, messages that are expected to be safely delivered at the remote nodes (according to the local views) will be safely delivered (depending on the global view) at a very high probability. Also, close view with the effect of reliable communications ensures precise prediction of node status and immediate notification of exceptions, since the predicting procedures are constructed through message passing.

Remember that 100% reliable delivery, even through a 100% reliable communication channel, is impossible, because the local view can only reflect, at best, the environment status in the near past due to the nature of message passing, and there is no way to guarantee that messages will be safely received by the destination nodes at the moment of transmission. Our simulations, which we describe later, demonstrate that the closer the relation among nodes, the more reliable message delivery is. That is, the more precisely a node can predict its local view of the status of all the other nodes, the less exceptions due to unexpected events regarding message deliveries.

Whether messages are safely delivered can be almost (at a very high probability) predicted, because of reliable communication and close view at the moment of message sending. Thus, it is unreasonable to sacrifice performance by waiting for acknowledgments if the extremely small probability of exceptions can be compensated for in some way. Highly reliable delivery in RDE, where the control flow continues immediately after the message sending without waiting for acknowledgments, leads to high performance for the targeted distributed computing tasks. One more important service in RDE is configuration exception control, which is proposed to complement highly reliable

delivery; this is discussed in the next section. Basically, the philosophy of the RDE model is highly reliable delivery plus configuration exception control.

### RDE Services

Two important services are supported in RDE: communication service and configuration service. Communication service supports reliable communication, and configuration service is for close view.

As we mentioned in the previous section, communication service uses the standard mechanisms of sequence number and time-out/retry to eliminate the possibilities of duplication, out of sequence and missing data packets in datagram communications. It supports three types of operations:

- *Multicast*. Send messages to a group of nodes. Nodes in the domain can be arbitrarily grouped by setting different channels. Grouping will be discussed with configuration service below.
- *End-to-end send*. Send message to one node. The two relevant function calls for this type are sendto and reply.
- *Multiread*. Read messages from multiple inputs. Two or more I/O ports can be created for different classes of communications. Each I/O port corresponds to a domain. Two or more domains can be specified in a distributed environment. Each domain defines a class of communication.

The major difference between RDE and traditional transport services is that the messages are assumed to be safely received by the destination nodes at the moment they are sent, and control is immediately returned to the caller. This is because messages are safely delivered with very high probability, which we explain in the "Simulation" section. The difficulties of highly reliable delivery of messages are solved by configuration exception control.

Configuration service supports the following:

- *Configuration exception control*. The idea is whenever RDE senses changing of configuration or unexpected conditions of message delivery, it notifies the higher level layer with a configuration exception. The service protocol guarantees that the related nodes will be notified in time *t* after a message delivery is initiated if the related exception occurs. The notification procedure first enters exception events into a global event queue, then generates a signal that invokes a handler routine.
- *Grouping*. Nodes can be grouped by setting channels. One node might become a member of multiple groups by setting two or more channels. A node can release group membership by unsetting the channels.
- *Dynamic configuration*. Nodes can come or go without affecting the whole system. This is very important in large-scale distributed systems that disallow turning off all the nodes in order to reconfigure the environment.

### Coupled Relation and Idealized RDE

In order to measure the degree of reliability of a distributed environment, the coupled relation is introduced as a function of how closely nodes must interact. The "more closely" the distributed nodes interact, the more reliably the distributed computing tasks are achieved. In other words, the coupled relation is used to quantify the reliability of a message passing-based distributed environment, where the reliability lies between closely-coupled and loosely-coupled with respect to message passing rather than concurrent processing. That is, we try to quantify the unreliability due to the deficiency of message passing. A closely-coupled distributed environment, where nodes communicate with each other through shared memory, can be considered 100% reliable with respect to message passing.

Two versions of coupled relation have been defined: as the coefficient of statistical correlation between a local view and the global view, and as the probability that a local view matches the global view. In most cases, there is a monotonically increasing relation between the coefficient of correlation and the probability. We simply use the second version as our experimental coupled relation for the simulations. This means that the coupled relation specifies how precisely a node's local view predicts the status of other nodes. If the coupled relation is equal to 1, we call it a closely coupled relation. In our RDE model, where the difficulties of highly reliable delivery of messages are solved by configuration exception control, a higher coupled relation that more precisely predicts node status will reduce the cost of exception handling. A coupled relation can therefore be considered a hit ratio where messages are safely delivered to destination nodes (as expected when sending messages).

Many parameters like message sending rate, node failure rate, node recovery rate, predicting algorithms, and data missing rate affect coupled relation; these are discussed in the "Simulation" section.

An idealized RDE is a distributed environment with the following conditions:

- *Closely-coupled relation*. This is the strongest version of the close view discussed above. The relation exists in distributed environments if all local views in active nodes are consistent with the global view at any time when message deliveries are initiated during the lifetime of an environment. The closely-coupled relation makes loosely-coupled distributed systems look like they share a portion of memory, the configuration bits, the major feature of closely-coupled distributed systems. This is the reason we call this the closely-coupled relation.
- *Immediate effect*. This guarantees that the status of the related nodes remains unchanged during the transmission of messages. This means that messages are received by the destination nodes instantaneously upon sending out the messages.

Idealized RDE guarantees 100% reliable delivery of messages.

It is impossible for idealized RDE to exist in a message passing system due to the nature of message passing: transmission time is required. The goal of this article is not to

achieve idealized RDE but instead to build a framework for distributed systems based on an architecture of highly reliable delivery plus configuration exception control; these services are supported in RDE. This framework makes it possible to achieve an almost idealized RDE.

### *Distributed Predicting Algorithms*

A distributed predicting algorithm is used in RDE in order to maintain local views. In the next sections we present three algorithms.

#### Global-Local Views Comparison (GL)

The information defining the global view can be collected approximately by receiving data/control packets from remote nodes. Local views are updated as follows:

- When a new node comes up or a node recovers from failure, send out control packets saying "I am up" to all nodes, and wait for acknowledgments, denoted by ACKs, from all nodes, within a time-out slice. This turns on the configuration bits corresponding to the nodes that the ACKs were received from, turning the rest of the configuration bits off.
- When a sending function is invoked, send data packets to the destination nodes and send control packets to all other nodes, and wait for ACKs from all active nodes within the time-out slice; turn off the configuration bits corresponding to the nodes from which expected ACKs are not received and turn on the configuration bits corresponding to the nodes from which unexpected ACKs are received. A configuration exception signal is generated whenever the local view is changed.
- When a packet is received, no matter whether it is a data packet or a control packet (excluding ACK), send back an ACK; if the configuration bit corresponding to the node that the packet is received from is off, turn it on. It would be unnecessary to send any packets to the nodes which the configuration bits indicate as inactive nodes, if the missing rate of packet transmission is equal to zero, where no unexpected ACKs that respond to control packets will possibly come back since all active nodes are noticed when one node comes up.

#### Local-Local Views Comparison (LL)

This algorithm inherits all features from the first algorithm except that the configuration bits are sent out with the packet to active nodes. Each receiving node compares its configuration bits with those received, and then turns off its configuration bits where an inconsistency occurs. The reason for turning off the relevant configuration bit instead of turning it on is because RDE communication service ensures reliable communication (low data missing rate). A node changing from inactive to active can broadcast a control packet, but a node changing from active to inactive cannot broadcast any more. The small probability of missing data ensures that almost all inconsistent bits that are on correspond to inactive nodes.

#### Enthusiastic-Correction (EC)

This algorithm inherits all features from the second algorithm. The node that turns the bits on when eventually receiving unexpected ACKs or turns the bits off when not receiving expected ACKs within the time-out slice sends out its configuration bits to all active nodes after updating. That is, the first node that notices the failure of a node informs all the other active nodes. An active node might receive a notice of the failure of itself due to packets missing in transmission. The node has to immediately broadcast a message "I am still alive!" to correct the inconsistency between local view and global view.

There are some other algorithms that could alternatively be used in RDE. For example, control messages could be periodically sent out in order to collect the most recent global view.

## Simulation

The purpose of the simulation is to understand the behavior of the coupled relation, and the performance and reliability associated with a coupled relation. The coupled relation indicates the reliability of message passing in a distributed environment based on our architecture. The performance of distributed computing tasks depends heavily on the reliability, which is indexed by the coupled relation of the environment; distributed computing tasks on a higher coupled relation environment incur lower exception handling costs.

The simulation assumes a distributed computing task running on an $N$-node environment where nodes interact with others only through broadcasting messages. The messages received by a node are independent with respect to the messages sent out by the node (independence assumption[13]). The time that it takes messages to be transferred between any possible pairs of nodes is assumed the same and small. Five environmental factors are used as the parameters in the simulation.

- *Distributed predicting algorithms*—GL, LL, and EC—provide different degrees of consistency between local views and the global view.
- *Node failure rate* $\lambda$ is the probability that an active node fails in a time unit. Assume $\lambda$ is a Poisson distribution.
- *Node recovery rate* $\mu$ is the probability that an inactive node recovers from failure within a time unit. The reciprocal of node recovery rate is the average time that a node stays in the failure state. Assume $\mu$ is a Poisson distribution.
- *Message sending rate* $\nu$ is the number of message passing operations invoked by an active node within a time unit. Assume $\nu$ is a Poisson distribution.
- *Data missing rate* $R_m$ is the probability that data packets are missing or cannot be delivered within a certain time slice. A fixed rate is assumed for each round of simulation.

The data missing rate is a complex of all possible transmission incompletions such as data missing due to physical

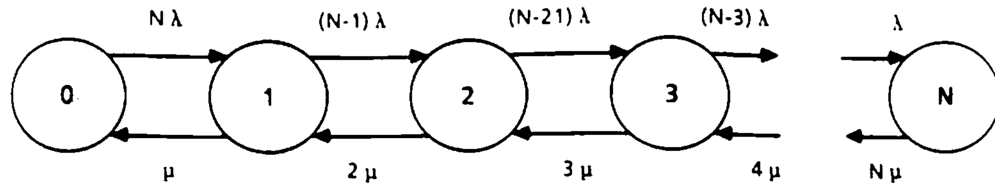



*Fig. 2. State diagram for node failure/recovery model — M/M/∞/∞/M.*

link failure, partition [14], or insufficient time-out slice. All transmission incompletions will be caught by the standard mechanism, *k*-retry/time-out. This is the reason we did not use time-out slice as a parameter, since this is more general and simple.

The model for node failure and recovery is actually a birth-death queuing system, M/M/∞/∞/M [15] (i.e., finite customer population and infinite number of servers). See Figure 2. The customer population (the number of the nodes) is $N$. A state $K$ is the state with $K$ nodes in failure status. The probability of going to state $K-1$ from state $K$ is $K \times \mu$, if $K>0$. The probability of going to state $K+1$ from state $K$ is $(N-K) \times \lambda$, if $K<N$. The number of message sendings in a time unit for state $K$ is $(N-K) \times \nu$, which plays an important role in RDE to approximate local views to the global view.

Basically, two stages of simulations were conducted. The first stage was to understand how the coupled relation is affected by the environmental factors, and the second stage was to understand how the system performance is affected by the coupled relation. Let *Cr* refer to the statistical value of coupled relation, which is between zero and one.

### *Coupled Relation with Different Environmental Factors*

In this section, we show how environmental factors—node failure rate ($\lambda$), node recovery rate ($\mu$), message sending rate ($\nu$), different distributed predicting algorithms (GL, LL, EC), and data missing rate ($R_{md}$)— affect *Cr*. Two utilization factors for the simulation are defined as follows:

- $\rho_m = \frac{\mu}{\nu}$: the ratio of node recovery rate to message sending rate.
- $\rho_r = \frac{\lambda}{\mu}$: the ratio of node failure rate to node recovery rate.

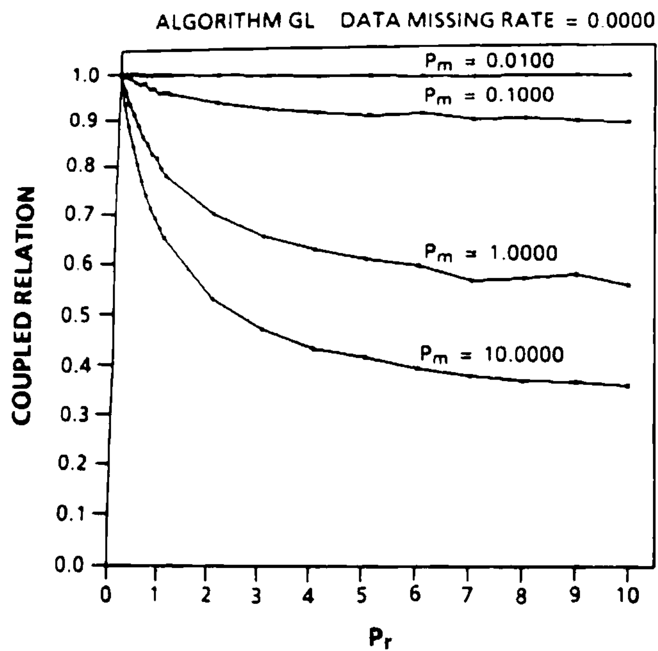
Generally, the coupled relation is approaching one, while $\rho_m$ and $\rho_r$ are close to zero. In Figure 3, we show the relation between *Cr* and $\rho_m$ and the relation between *Cr* and $\rho_r$, respectively. We assume algorithms GL and $R_{md} = 0$ in both simulations.

$\rho_m$ carries the information concerning message sending during the period of node failure. If the average number of failure nodes is $K$ and the number of nodes in the environment is $N$, then the average number of messages broadcast, $M$, from the time a node fails to the time the node recovers is $\frac{(N-K) \times \nu}{\mu}$, which is $\frac{(N-K)}{\rho_m}$. $K$ is dependent on $\rho_r$; $K$ is small while $\rho_r$ is small, which implies the high *Cr*.

Message broadcasting during the period of node failure plays an important role in correcting the inconsistency between local views and the global view, which is caused by the failure of nodes. If $M$ is much larger than $1$, only the nodes broadcasting the first few messages (depending on the predicting algorithms, for EC, only the one node) suffer the incorrectness of local views due to the node failure, and all the nodes that broadcast the following messages take advantage of correct prediction of local views, which are delivered without exception. Thus, the lower $\rho_m$ implies the higher $M$, which implies the higher *Cr*. If $M$ is small, the effect of the control message "I am up" broadcasting when a node recovers from failure is significant, most likely causing the local views of all active nodes to be corrected before any data packets are broadcast. Conclusively, in order to get a high *Cr*, it is important that $\rho_m$ and $\rho_r$ stay small, and the factor of $\rho_m$ affects the coupled relation more than $\rho_r$ does. That is, increasing $\rho_m$ has a more significant effect on decreasing the coupled relation than does increasing $\rho_r$. Next, we will compare the different predicting algorithms in the cases that $R_{md}$ is fixed to zero and $R_{md}$ is changing.

In Figure 4, we show *Cr* for the three distributed predicting algorithms where $R_{md}$ is zero. Two simulations are conducted: i) $\rho_m$ is 0.1 and $\rho_r$ is from 0.01 to 10.0; and ii) $\rho_r$ is 0.1 and $\rho_m$ is from 0.01 to 10.0. We can see, generally, EC has a higher *Cr* than LL in most cases, and LL has a higher *Cr* than GL. That is, when the data missing rate is zero, algorithm EC is better than algorithm LL, which is better than algorithm GL. The reason is obvious: LL requires sending out packets to active nodes with local views and GL does not, and EC requires informing all nodes when the first node discovers any node failure and/or node recovery and LL does not. It is relatively more expensive for EC to maintain a higher *Cr* than it is for LL, which is more expensive than GL.

In Figure 5, we show *Cr* for the three algorithms in the case where $R_{md}$ is changing. In the simulation where $\rho_r$ and $\rho_m$ are both 0.01, the relation that EC is better than LL, and LL is better than GL no longer holds as soon as $R_{md}$ is greater than 0.0001. Algorithms EC and LL have a low *Cr* when $R_{md}$ is high, where GL still has a comparatively high *Cr*. In the simulation that $\rho_r$ and $\rho_m$ are both 0.1, EC is better than LL, and LL is better than GL when $R_{md}$ is smaller than 0.001, and then, when $R_{md}$ is greater than 0.001, GL becomes better than both LL and EC. This is because the information that EC and LL need to inform all other nodes is very likely incorrect since $R_{md}$ is high. GL affects local views



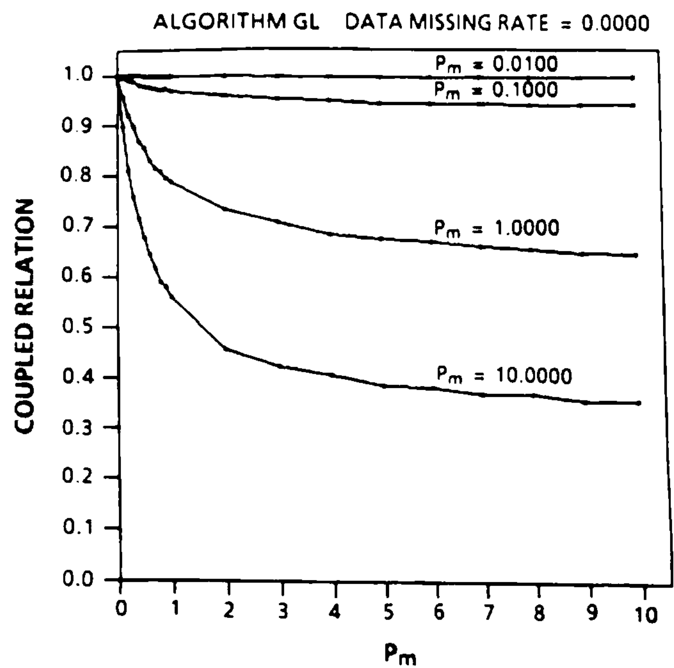
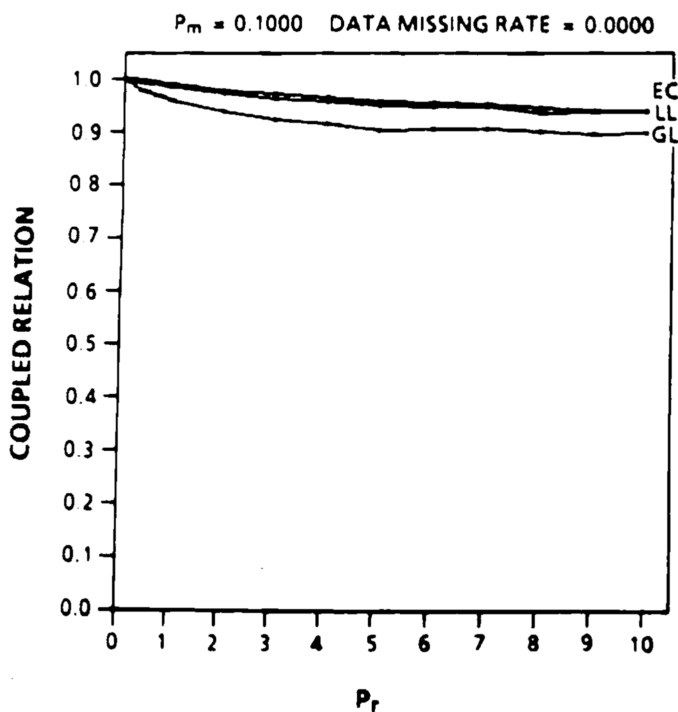




*Fig. 3. Coupled relation and $\rho_m$, $\rho_r$*



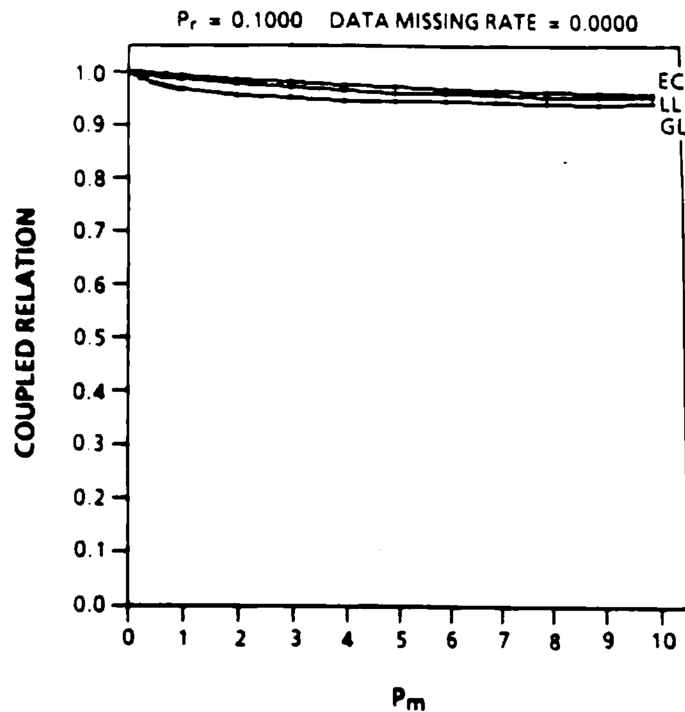
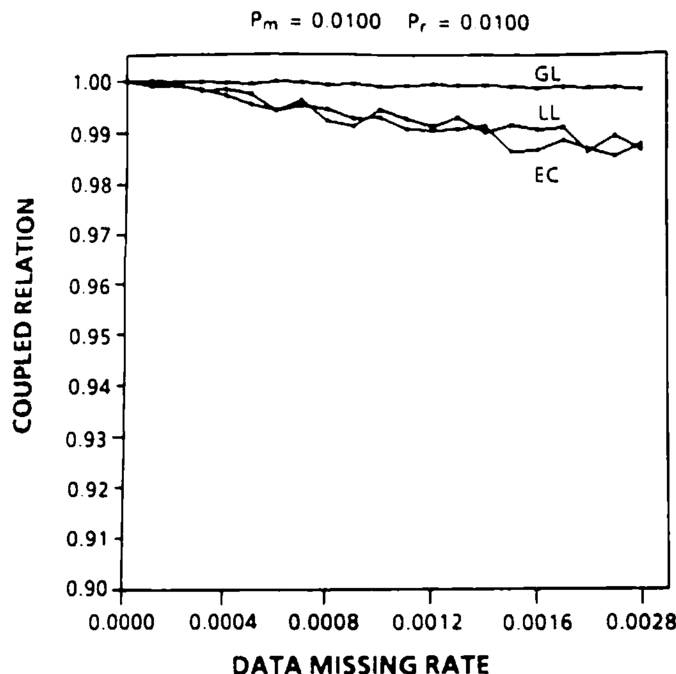




*Fig. 4. Coupled relations for predicting algorithms when $R_{ms}$ is zero.*



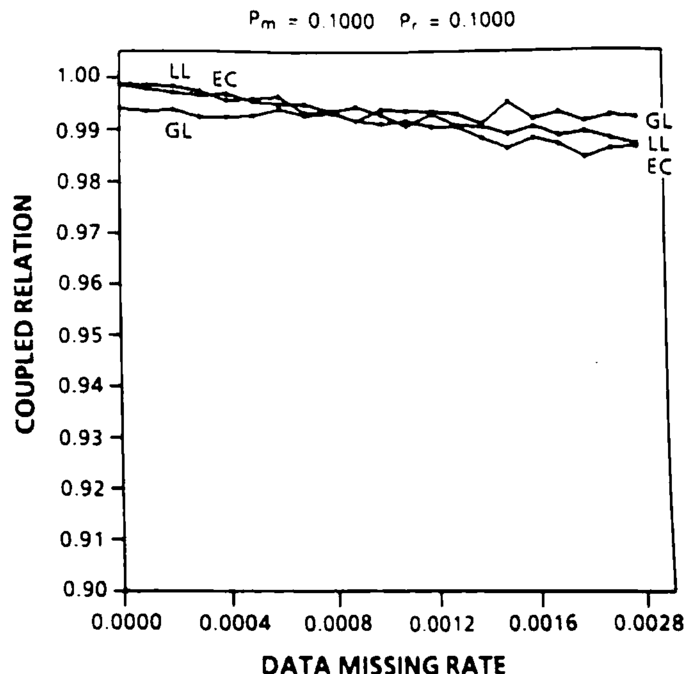




*Fig. 5. Coupled relations for predicting algorithms when $R_{mi}$ is changing.*

only by the comparison with the global view rather than from rumors. Conclusively, when $R_{mi}$ is high and $\rho_r$ and $\rho_m$ are small, GL is better than LL and LL is better than EC. When $R_{mi}$ is small and $\rho_r$ and $\rho_m$ are high, EC is better than LL and LL is better than GL.

Figure 6 shows that when $R_{mi}$ is very high, GL is much better than LL and EC and much more stable (almost linear).

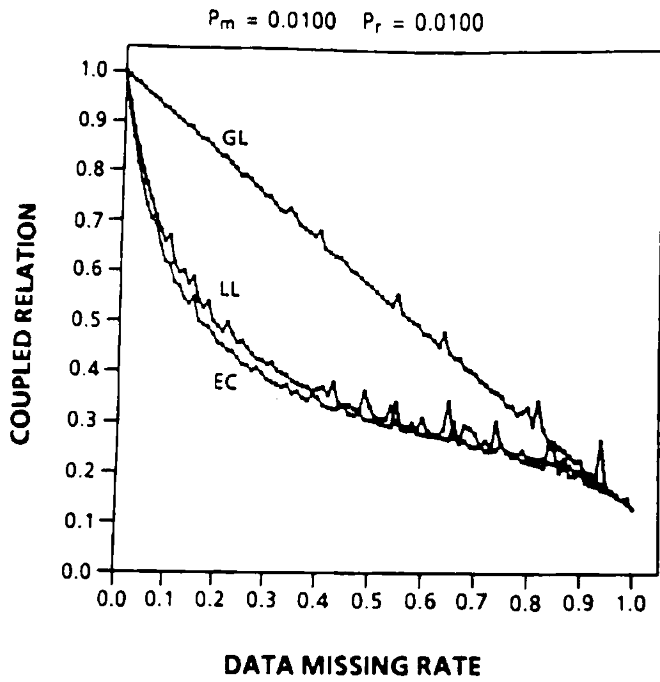
### *Performance with Coupled Relation*

In this stage of simulation, we show how *Cr* affects the environment performance. Instead of measuring the environment performance directly, the exception cost is measured in the simulation. An exception is signaled when a message, sent to a node that is indicated active in the local view, is not safely delivered due to incorrect local view, incomplete transmission, or the failure of remote nodes when transmitting. Incorrect local view is indexed by the low *Cr*. Incomplete transmission is indexed by data missing rate $R_{mi}$ and failure of remote nodes is indexed by node failure rate $\lambda$. The cost of an exception includes the cost that RDE needs to detect the incorrect local view and notify the higher level layer with a signal, and the cost of executing the application-specific compensation function (exception handler routine). We thus assume the exception cost is much more expensive than the cost of message passing. The overall exception cost for a distributed computing task can be considered the same as the number of exceptions, if we assume the cost from all exceptions are the same. In Figure 7, the higher *Cr* causes the lower exception cost. More importantly, every curve is almost linear and the relation between the starting point (when *Cr* is 1.0) of each curve and the data missing rate $R_{mi}$ is also almost linear. This makes the prediction of the exception cost according to the coupled relation very easy.

## View Section

Our view section model is a programming framework for constructing distributed computing tasks on top of RDE. View sections protect against changes to the imagined shared memory, the global view, in the same way that critical sections protect against change of real shared memory. Certain differences exist, e.g., for critical sections, shared variables can be locked against being further accessed until they are unlocked. In contrast, the change of the global view due to the failure of nodes is totally out of control, so there is no way to prevent the global view from changing. The philosophy is that a view section, which defines a period of time and a sequence of instructions, is declared as a protected section during which the global view is desired to remain the same. If the global view does change during the protected section, a compensation function, defined in the beginning of the view section, is invoked by a notification generated from the underlying RDE. A compensation function behaves as a special form of exception handler. The service supported by the underlying RDE is configuration exception handling, as mentioned above.

Each view section begins with the statement `Begin_View_Section`, and may or may not end with the statement, `End_View_Section`, as illustrated in Figure 8 We call them close-type view section and open-type view section, respectively, to distinguish the static characteristic



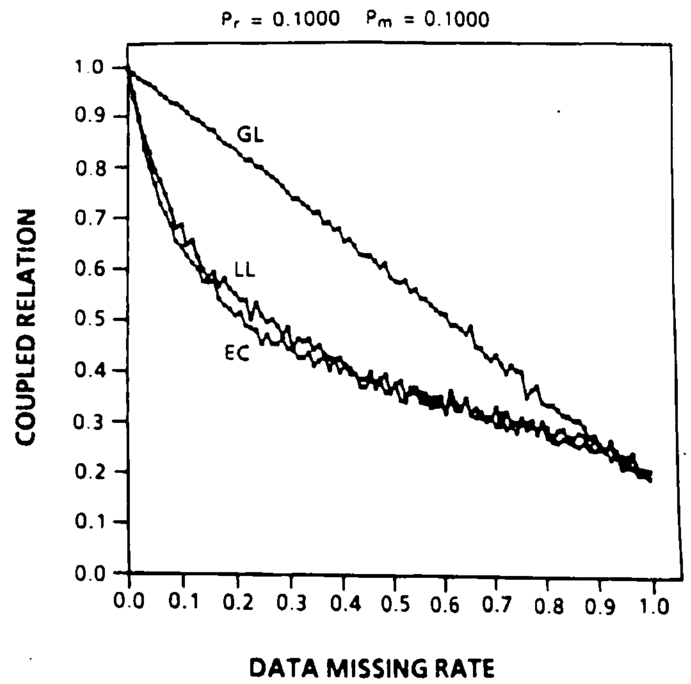
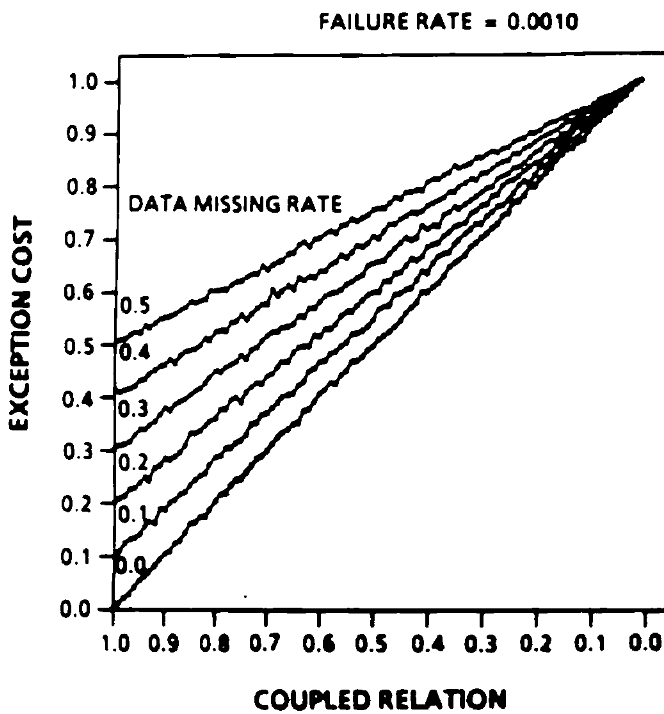




*Fig. 6. Coupled relations for predicting algorithms when $R_{ms}$ is changing.*



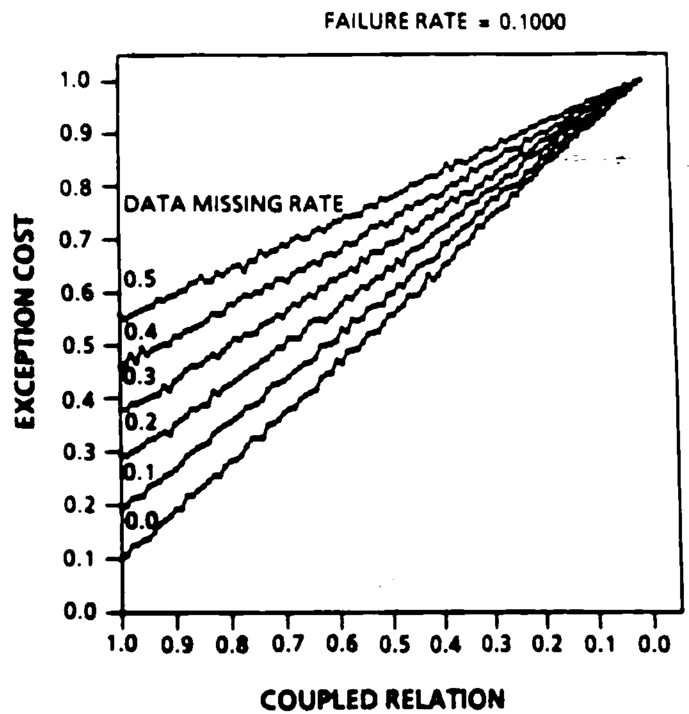




*Fig. 7. Exception cost affected by Cr.*

```
Open-Type View Section

vsec := Begin_View_Section(group,timeslice,function)
. . . .

Close-Type View Section

vsec := Begin_View_Section(group,timeslice,function);
. . . .
End_View_Section(vsec);
```

*Fig. 8. View sections.*

of programming structure. At run-time, a view section may end with the statement End_View_Section or end in a given time slice. We call them code-bound view section and time-bound view section, respectively. An open-type view section must be a time-bound view section, but a close-type view section does not have to be a code-bound view section—it may end due to code or due to time.

The statement Begin_View_Section initializes a view section. The first parameter group specifies a set of participant nodes. The second parameter timeslice is the maximum time period in which the tasks executed in the view section are expected to be accomplished. Programmers can set the time slice themselves, or use a time estimation function, provided by the underlying RDE, to get the time slice. Alternatively, they can leave the problem to RDE by setting DEFAULT. The third parameter, function, is an exception handler for when exceptions arise during the view section. Because a computing task may be involved in more than one view section at the same time, vsec acts as a handle that uniquely identifies a particular view section. The handle vsec is very important when nested view sections or overlapped view sections are allowed. For example, an open-type view section can be inside a close-type view section.

The statement End_View_Section terminates the view section immediately. If given time period is consumed before the statement End_View_Section is executed, the view section is forced to end by implicitly invoking an RDE procedure that checks the status of the distributed environment and, if necessary, invokes the compensation function. Then the program continues to execute the current statement if the view section is open-type, or jumps to the next statement following End_View_Section if close-type.

Another important statement is:

Update_View_Section (vsec, timeslice, function)

which reinitializes view section vsec and changes the time slice and the compensation function if necessary. If programmers do not want to change the parameters, they can set the parameters to SAME. Parameter group is unnecessary here. This statement can also be used to end a view section if the time slice is set to zero. The possible usage is that it is called from the compensation function to end the view section when a fatal exception arises.

One typical pattern of view section, shown in Figure 9, is sending a message to all nodes in group and then receiving a message from each active node. In fact, the mechanism of positive acknowledgment plus time-out existing in many communication layers describes some aspects of this pattern of view section. The view section model not only provides a clear structure to programmers, but also provides a precise configuration exception handling. A distributed computing task constructed by using view sections on top of RDE is aware of the exception that a node recovers as well as the exception that a node fails.

Nested view sections and overlapping view sections are permitted in our model. Many problems arise due to the time-out slices, but these are outside the scope of this article. In the next sections, we give two examples using view sections.

### *Example 1: Summation of Distributed Data*

This example does not illustrate the feature of reliabilit when using view section. It illustrates only how to use viev section on top of RDE. (See Figure 10.) Example 2 describe how reliability is achieved by using the view section model.

```
vsec := Begin__View__Section(group,timeslice,function);
send(group, message);
for (node i in group is active) receive(message);
End__View__Section(vsec);
```

*Fig. 9. Typical view section pattern.*

```
Procedure Sum;
begin
  for (node NodeId in group) MY__X[NodeId] := 0;
  S := 0;
  vsec := Begin__View__Section(group,stime,Adjust__Sum);
  multicast(group, "send back X value");
  for (node NodeId in group is active)
  begin
    receive(X);
    S := S + X;
    MY__X[NodeId] := X;
  end;
  End__View__Section(vsec);
  Sum := S;
end;

procedure Adjust__Sum;
begin
  Get__Exception(NodeId, NodeStatus);
  if (NodeStatus = INACTIVE) then
  begin
    S := S - MY__X[NodeId];
    MY__X[NodeId] := 0;
  end
  else if (NodeStatus = ACTIVE) then
  begin
    send__to(NodeId, "send back X value");
    Update__View__Section(vsec,stime,SAME);
  end;
end;
```

*Fig. 10. Summation of distributed data.*

A system has $N$ nodes, which are each randomly active or inactive. Each node has a variable $X$ that changes in value from time to time. A designated node executes a function that returns the sum of the $X$ values for all active nodes. The multicast function is supported by RDE. Two functions are presented below to solve the problem. The main function is Sum, which broadcasts a message to all active nodes and then waits to receive the $X$ value from each active node.

After it initializes the array $MY_X$ *and* $S$, Sum begins a view section by executing the statement Begin_View_Section, which defines a time-out slice, stime, and a compensation function, Adjust_Sum. Then the Sum function multicasts a request to all active nodes and waits for the $X$ values from all active nodes. It adds each received $X$ value to $S$ until values have been received from all active nodes.

The compensation function, Adjust_Sum, will be invoked as an exception handler routine whenever the global view is changed. The function Get_Exception will return the node whose status changed to cause the exception and the status of the node (i.e., NodeId and NodeStatus). If the status of the exceptional node changes from active to inactive, then it will subtract $MY_X$'s value of node NodeId from the sum and set $MY_X$'s value of node NodeId to zero. If the status of the exceptional node changes from inactive to active, it will send a request message to the exceptional node and update the view section by resetting a new time-out stime, which ensures that the exceptional node has enough time to send back its $X$ value. The compensation function of the view section is still bound to Adjust_Sum (i.e., SAME).

*Example 2: Reliable Resource Redistribution*

A distributed environment has $N+1$ nodes, $node_0$, $node_1$,..., $node_N$, which may be active or inactive. $Node_0$ is the leader that was previously elected by all the nodes. Each node has several resources that might be allocated by a local process. For $node_i$, the set of available resources is $R_i$. The leader, $node_0$, invokes a task of resource redistribution upon a request from another node that has consumed all its available resources. The leader broadcasts a message to ask other nodes to relinquish their available resources. After the leader receives all relinquished resources, $R_1 + R_2 + ... + R_N$, it reassigns resources so that the set of available resources for node $i$ is $Q_i$. Then the leader sends $Q_1$, $Q_2$,..., $Q_N$ to $node_1$, $node_2$,..., $node_N$, respectively.

The problem is that every node, including the leader, might fail unexpectedly during the process of resource redistribution. We do not want to lose or duplicate resources due to the failure of regular nodes (i.e., not the leader). We also do not want to swallow resources due to the failure of the leader; this might block or dramatically slow down the whole system. We do not address the reassignment problem, but we assume it takes time to complete this task.

The main function is Resource_Redistribution, which divides into three blocks as shown in Figure 11. In the first block, it initializes a view section with the compensation function Check_Total. It multicasts a message REQUEST to ask all nodes to give up and send back their available resources $R_i$, and then waits for all resources to be relinquished. After it receives the available resources from all active nodes, it reinitializes the view section with compensation function Check_Fail and goes to the second block. In the second block, it reassigns resources into $Q_i$, and sends the $Q_i$ to nodes. One important consideration is that the leader has to ask nodes to lock resources $R_i$ before they get $Q_i$ back, and if a node fails before getting $Q_i$, it should consider $R_i$ as the available resources when it recovers from failure. This prevents the received resources from premature allocation to local processes until the leader makes sure that the redistributed resources have been safely received by all nodes. If $node_i$ fails during the second block, i) all resources that were already distributed have to be canceled; ii) the resources received from $node_i$ in the first block have to be discarded so that $node_i$ can consider $R_i$ as its available resources after it recovers from failure; and iii) reassignment of resources has to be done again and then the leader redistributes the sets of resources. The view section ends at the end of the second block. In the third block, not inside the view section, it broadcasts a message OK to unlock the resources. Message OK indicates the leader knows all nodes have received the newly assigned resources $Q_i$.

Two compensation functions are used in the view section: Check_Total in the first block and Check_Fail in the second block. Check_Total is almost the same as the compensation Adjust_Sum in the previous example "Summation of Distributed Data" except that resources are used here. In the second block, Check_Fail is the compensation function which, when some $node_i$ fails, asks all the nodes that already received resources $Q_i$ to give them up, and restarts the second block again. Function Set_Resume is used to jump gracefully to the beginning of the second block, labeled REASSIGN. We do not care about the case that nodes are restored from previous failures, because it is too late to reassign the resources for the "coming up" node. It has to wait until the next round of resource redistribution.

From the view point of a regular $node_i$, several rules are followed:

- $Node_i$ considers $R_i$ as its available resources in the following cases:
  a. $Node_i$ does not receive $Q_i$ within a given time slice, after it sends out $R_i$. When time-out occurs, the leader is assumed to have failed.
  b. $Node_i$ fails after it gives up $R_i$ and before it receives its newly assigned resources $Q_i$.
  c. $Node_i$ does not receive an OK message from the leader within a given time slice. The leader is assumed to have failed. The OK message is sent out by the leader, when the leader makes sure all nodes received newly assigned resources $Q_1$, $Q_2$,..., $Q_N$.

  This is to protect resources from being swallowed by the leader, if the leader node fails after it receives part or all of the resources.
- $Node_i$ considers $Q_i$ as its available resources, if it receives an OK message after it received $Q_i$ from the leader.
- $Node_i$ considers $R_i$ and $Q_i$ as its possible available resources, if it fails after it receives $Q_i$ and before it receives an OK message. A checking procedure, which checks whether $R_i$ or $Q_i$ is its available

```
procedure Resource__Redistribution;
begin
  for(node NodeId in group) R[NodeId] := INVALID;
  vsec := Begin__View__Section(group,stime,Check__Total);
  multicast (group,REQUEST);
  for (node NodeId in group is active)
    receive (R[NodeId]);
  Update__View__Section(vsec,stime,Check__Fail);
REASSIGN:
  reassign__resources(R[ ], Q[ ]);
  for (node NodeId where R[NodeId] is valid)
    send__to(NodeId, Q[NodeId]);
  End__View__Section(vsec);
  multicast (group, OK);
end;

procedure Check__Total;
begin
  Get__Exception(NodeId, NodeStatus);
  if (NodeStatus = ACTIVE) then
  begin
    send__to(NodeId, REQUEST);
    Update__View__Section(vsec,stime,SAME);
  end;
end;

procedure Check__Fail;
begin
  Get__Exception(NodeId, NodeStatus);
  if (NodeStatus = INACTIVE) then
  begin
    for (node NodeId to which Q[NodeId] has been sent)
      multicast (group,DISCARD);
    R[NodeId] := INVALID;
    Update__View__Section(vsec,stime,SAME);
    Set__Resume(REASSIGN);
  end;
end;
```

*Fig. 11. Reliable resource redistribution.*

resources, will be invoked when $node_i$ recovers from failure; that is the procedure. $Node_i$ broadcasts a message to ask whether $Q_i$ is a valid resource set or not, where we can use version number of $Q_i$ to verify it. If $node_h$, which was the leader when $node_i$ failed, says "no", it uses $R_i$ as its available resources. If any node says "yes", it uses $Q_i$ as its available resources. Otherwise, it waits until a node comes up, and repeats the whole procedure. This is to protect resources from being duplicated or lost.

A state diagram is presented in Figure 12 to describe the behavior of regular nodes.

## Conversations

Peterson's conversations [16] are another IPC abstraction that attempt to achieve reliability in distributed computing by incorporating a notion of view. In a conversation, however, a view is a context graph giving the partial ordering of all past messages. As with our RDE, the global view is the truth and the local view is what distributed nodes know about the truth. Distributed nodes sense the truth through message passing, and the difference between the global view and a local view is due to the unreliability of message passing. That is, a node may not be aware of the entire context due to missing or out of order messages, as well as node failures or network partitions. Each message is passed with the entire context graph known by the sender, so the receiver has the opportunity to update its own and/or the sender's view of the truth with any messages included in one but not the other.

Peterson's conversations and our RDE have the following features in common:

- *Global view*. There exists a global entity, the truth, which might change from time to time. In the conversation IPC abstraction, it is the context graph. In our model, it is the (imaginary) global configuration bits.
- *Local view*. Every node has a view that represents its knowledge about the global view. Local views are updated incrementally through message passing and are rebuilt after node failure or network partition. In conversations, it is participant $p$'s view of the context graph. In our model, it is the local configuration bits.
- *Knowledge basis*. Distributed nodes behave according to their local views, and local views and operations

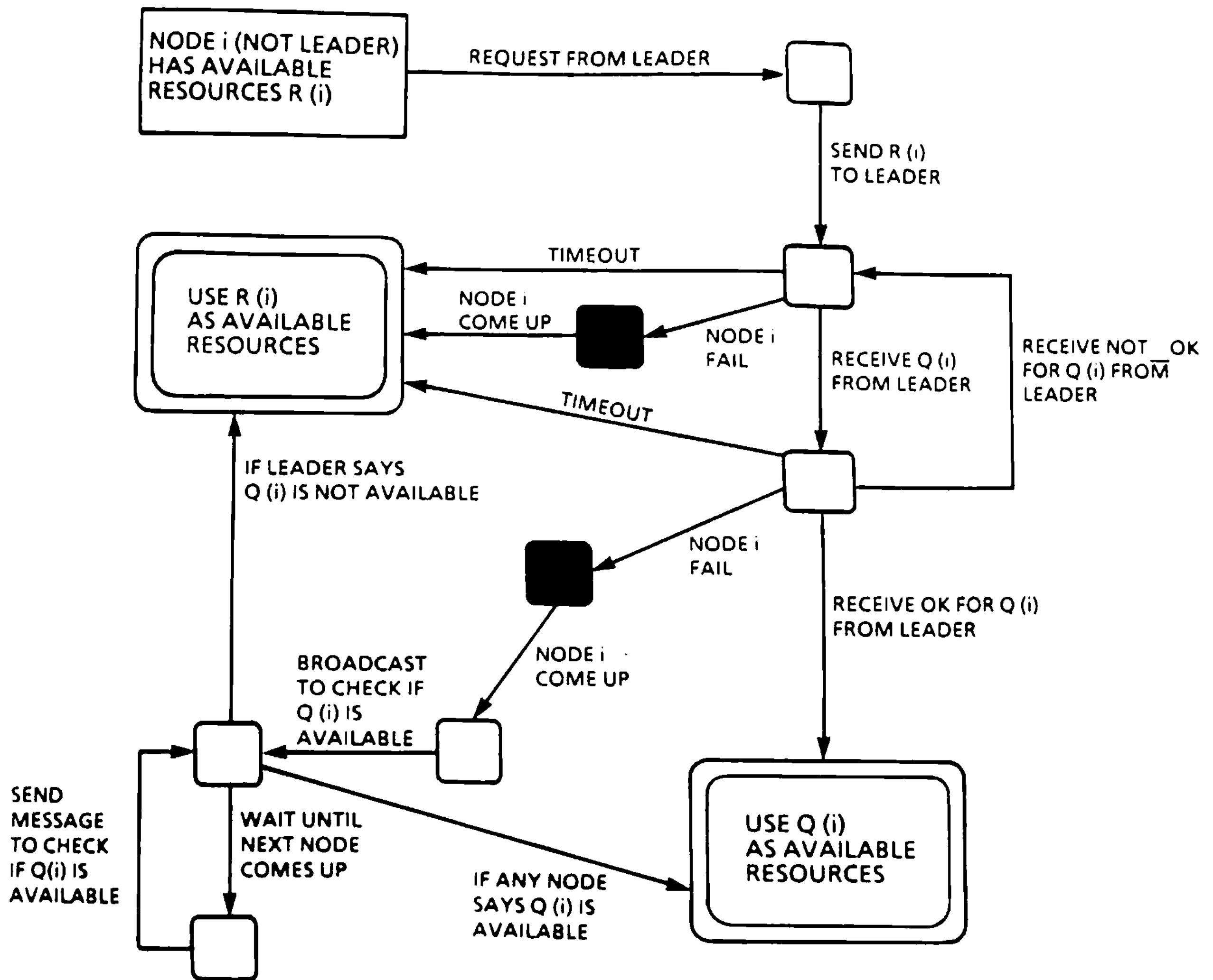



*Fig. 12. State diagram.*

are mutually affected. In conversations, the operations are the standard send() and receive(). Our model also provides sendto() and multicast().

Peterson attempts to achieve ordering of messages for group communication, whereas we try to attain safe delivery of messages for group communication. This difference in goals explains the differences between views. The dynamic views used in conversations preserve context information, while the (conceptually) static views used in RDE preserve configuration information. Conversations support ordered broadcast while RDE protects against node failure.

Thus, conversations and our model are really complementary. It would be nice to implement conversations on top of RDE, using view sections to build the component of conversations that protects against node failure. This would solve certain problems of conversations, such as nodes expecting an acknowledgment from a failed process and lack of reliable broadcasts. It would also be easy to use view sections to rebuild the context graphs when the system recovers from node failure or network partition.

## Conclusions

We propose a Reliable Distributed Environment (RDE) among large groups of nodes to ensure the reliability of complicated communication patterns, as virtual circuits are already applied to pairs of nodes to ensure the reliability of simple communications. RDE serves as a basic communication environment to handle large-scale and/or intelligent distributed systems. RDE provides reliable communication services and configuration services that do not exist in traditional communication environments.

We characterize the reliability of a distributed environment by its coupled relation, which is based on differen predicting algorithms, node failure rate, node recovery rate message sending rate, and data missing rate. Our simula tion results clearly illustrate how the coupled relation af fects the performance of a distributed environment.

We have demonstrated the utility of our view sectio model by our reliable resource redistribution example. W believe this model, which supports a high level of abstrac tion for handling low-level environmental changes, wi

prove to be a good programming framework for constructing reliable distributed computing tasks.

We expect our network architecture, with view section as the top layer, RDE in the middle and datagram communication at the bottom, to become increasingly significant due to the movement towards large-scale distributed systems and intelligent distributed applications.

## Acknowledgments

This article is an expansion of a paper with the same name that appeared in the proceedings of the *1987 Symposium on Simulation of Computer Networks*, Colorado Springs, CO, August 1987, pp. 11–22. This research is supported in part by grants from AT&T Foundation, IBM, and Siemens Research and Technology Laboratories, in part by the New York State Center of Advanced Technology—Computer and Information Systems and by the Columbia University Center for Telecommunications Research, and in part by a Digital Equipment Corporation Faculty Award.

## References

[1] H. S. Stone, *Computer Science Series: Introduction to Computer Architecture*, SRA, Chicago, 1980.

[2] K. Ravindran and S. T. Chanson, "State Inconsistency Issues in Local Area Network-Based Distributed Kernels," *Proceedings of Fifth Symposium on Reliability in Distributed Software and Database Systems*, Los Angeles, CA, Jan. 1986.

[3] W. L. Gentleman, "Message Passing Between Sequential Processes: The Reply Primitive and the Administrator Concept," *Sofware—Practice & Experience*, vol. 11, pp. 435–466, May 1981.

[4] L. Svobodova, "File Servers for Network-Based Distributed Systems," *ACM Computing Surveys*, vol. 16, pp. 353–398, Dec. 1984.

[5] J. G. Mitchell and J. Dion, "A Comparison of Two Network-Based File Servers," *Communications of the ACM*, vol. 25, pp. 233–245, Apr. 1982.

[6] G. R. Andrews and F. B. Schneider, "Concepts and Notations for Concurrent Programming," *ACM Computing Surveys* , vol. 15, pp. 3–44, Mar. 1983.

[7] J. Eliot and B. Moss, "Nested Transactions: An Approach to Reliable Distributed Computing," Technical Report, Massachusetts Institute of Technology, Apr. 1981.

[8] B. W. Lampson, "Atomic Transactions," *Distributed Systems—Architecture and Implementation: An Advanced Course*, Springer-Verlag, Berlin, FRG, Chapter 11, 1981.

[9] A. S. Tanenbaum, *Computer Networks*, Englewood Cliffs, NJ: Prentice-Hall, Inc., 1976.

[10] J. Kramer and J. Magee, "Dynamic Configuration for Distributed Systems," *IEEE Transactions on Software Engineering*, vol. SE-11, pp. 424–434, Apr. 1985.

[11] S. Sechrest, "An Introductory 4.3 BSD Interprocess Communication Tutorial," Technical Report, Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA, 1986.

[12] S. J. Leffler, R. S. Fabry, W. N. Joy, and P. Lapsley, "An Advanced 4.3 BSD Interprocess Communication Tutorial," Technical Report, Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA, 1986.

[13] L. Kleinrock, *Communication Nets: Stochastic Message Flow and Delay*, NY: McGraw-Hill, 1964, Reprinted, Dover Publications, 1972.

[14] S. B. Davison, H. Garcia-Molina, and D. Skeen, "Consistency in Partitioned Networks," *ACM Computing Surveys*, vol. 17, pp. 341–370, Sept. 1985.

[15] L. Kleinrock, *Queueing Systems, Volume I: Theory*, NY: John Wiley & Sons, 1975.

[16] L. L. Peterson, "Preserving Context Information in an IPC Abstraction," *Proceedings of Sixth Symposium on Reliability in Distributed Software and Database Systems*, Kingsmill Williamsburg, VA, Mar. 1987.

**Wenwey Hseush** received his M.S. from Columbia University and his B.S. from National Taiwan University, and will become a Ph.D. candidate at Columbia University in Fall 1988.

Hseush is a Research Staff Associate of Computer Science at Columbia University. His research interests include distributed programming environments, object-oriented languages, distributed systems and network simulation.

**Gail Kaiser** received her Ph.D. and M.S. degrees from Carnegie Mellon University, where she was a Hertz Fellow, and her Sc.B. from the Massachusetts Institute of Technology.

Kaiser is an Assistant Professor of Computer Science at Columbia University. She received a Digital Equipment Corporation Incentives for Excellence award in 1986, and has been selected as a National Science Foundation Presidential Young Investigator for 1988.

Her research interests include programming environments, evolution of large software systems, appplication of artificial intelligence technology to software development and maintenance, software reusability, object-oriented languages and databases, and distributed systems.

IEEE NETWORK
THE MAGAZINE OF COMPUTER COMMUNICATIONS

July 1988 ISSN 0890-8044

Simulation and Analysis of Very Large Area Networks (VLAN) Using an Information Flow Model / *Jacob J. Wolf III, Biswadip Ghosh* 6
Performance Analysis of a Large Interconnected Network by Decomposition Techniques / *Jhitti Chiarawongse, Mandyam M. Srinivasan, Toby J. Teorey* 19
A Network Architecture for Reliable Distributed Computing / *Wenwey Hseush, Gail E. Kaiser* 28
Effects on Response Time Performance Using an Edge-to-Edge Protocol in an X.25 Packet Network / *Paul T. Brady* 45
MBRAM - A Priority Protocol for PC Based Local Area Networks / *Robert P. Signorile, James LaTourrette, Michael Fleisch* 55

---

Guest Editorial / *Mitchell G. Spiegel* 5
The IEEE Network Forum / *John Daigle, Univ. of Rochester* 60
Technology Perspective / *Eric E. Sumner, AT&T Bell Laboratories* 61
Information Infrastructure / *Vinton G. Cerf, Corporation for National Research Initiatives* 62
Open Systems Standards / *Hal Folts, Omnicom, Inc.* 63
Conference Calendar 65
Advertiser's Index 68

---

Director of Publications—*Stewart D. Personick*
Editor-in-Chief—*Harvey A. Freeman*
Publication Editor—*John N. Daigle*
Publisher—*Carol M. Lof*

Cover Photo: The Image Bank/Photographer: S. Hunt