# Distributed ML: Abstractions for Efficient and Fault-Tolerant Programming

Clifford Dale Krumvieda
Ph.D Thesis

93-1376
August 1993

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

# DISTRIBUTED ML: ABSTRACTIONS FOR EFFICIENT AND FAULT-TOLERANT PROGRAMMING

A Dissertation
Presented to the Faculty of the Graduate School
of Cornell University
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy

by
Clifford Dale Krumvieda
August 1993

DISTRIBUTED ML: ABSTRACTIONS FOR EFFICIENT AND FAULT-TOLERANT PROGRAMMING

Clifford Dale Krumvieda, Ph.D.
Cornell University 1993

Despite the availability, inherent parallelism, and potential fault tolerance of networked workstations and microcomputers, most programmers do not write distributed code. Those that do are often overwhelmed by the asynchrony, concurrency, and tricky failure behaviour inherent in such systems.

In this thesis, we describe the design and implementation of a new programming language called *Distributed ML.* Distributed ML provides a programming construct called a *port group* that hides the sources of complexity listed above and can be implemented efficiently. Port groups are intermachine multicast channels which provide membership and failure information to application programmers. Although inherently asynchronous, port groups guarantee the delivery of data sent through them and can order such data in several different ways, thereby providing many of the assurances of synchronous communication. Port groups are general-purpose communication abstractions that can be used to transfer information between machines, between processes on the same machine, and between threads within the same pro-

cess.

In this thesis, we demonstrate that efficient distributed programs—even highly available and fault-tolerant distributed programs—can be quickly developed, easily reasoned about, and properly coded in a well-designed high level programming language. First, we provide an implementation and description of port groups in the context of the *Concurrent ML* concurrent programming language, which is a superset of the *Standard ML* general-purpose programming language. Second, we introduce a formal theory for relating the membership and ordering properties of port groups. Finally, we argue that our implementation matches the formal specification.

# Biographical Sketch

Clifford Dale Krumvieda was born on December 30, 1967 in Portsmouth, Virginia. In 1973 he followed his parents to the Panama Canal Zone, and in 1977, Cliff moved to Fort Worth, Texas, where he lived the good life. After graduating from Richand High School in 1985, he studied Applied Mathematics at Texas A&M University and was graduated *summa cum laude* in 1989. In the fall of that year he entered the field of Applied Mathematics at Cornell University, and in 1990 he transferred to the Department of Computer Science. In May, 1992 he married Ms. Rebecca Anne Beu. He received his Ph.D. from Cornell in August 1993.

To Becky, my partner and soulmate during this life,

and to all those who cared for Cindy,

the York Steak House dishmachine.

# Acknowledgements

Several mortals have helped to construct the man who wrote this thesis. My parents, Ron and Linda Krumvieda, conceived and brought me into this world; I thank them for my wonderful childhood home and the loving support they still provide. I learned more about life from Sam Holland and the other employees at York Steak House than I did in twenty years of school, and I thank them for their friendship and the lessons we learned together. I owe most of my creative expressibility and political views to Fred Lister, chief architect of *The Story War*. Danny Williams, my perennial friend and competitor, forced me to hone my technical skills, and I thank him for sharing his wit and spontaneous style. One of the few things that I will miss in Ithaca are the Nomads; I will never forget their friendship, support, and sometimes weird Bible interpretations. In particular, I want to thank Mike Thompson for proving that getting older doesn't necessarily imply growing up.

Mrs. Agee, my eighth grade Algebra teacher, is the best teacher in the world and without her instruction I would not have pursued a technical degree.

I thank the founders of Texas A&M University for creating a school rich in tradition and respect for its own history. I had many good professors at A&M, and Dr. Kroitor, Dr. Bryant, and Dr. Johnson were the cream of the crop. Helen Doerr, Sue

Martin, and others at the Cornell National Supercomputer Facility introduced me to Cornell during a Research Experience for Undergraduates Program, and I appreciate their kindness. Dr. Northrup Fowler, Jamie Lawton, and Jennifer Skiddles supported me through two summers at Rome Laboratories in Rome, New York.

I appreciate all those with whom I have had useful, friendly, technical, and political discussions at Cornell, especially Mike Reiter, Ken Birman, Brad Glade, Dalia Malki, Robbert van Renesse, and Guerney Hunt. I especially thank the comments of my Special Committee, Bard Bloom, Keith Marzullo, Tim Teitelbaum, Richard Shore, and Robert Cooper. Bard in particular deserves my thanks and praise for smoothing out my sporadic and sometimes weird research attempts.

I am grateful for the financial support of Texas A&M University, Coors, the Department of Defense, IBM, and the Isis Project.

Last, but certainly not least, I want to thank my wife Becky, for her undying love, and God, for providing me with a way to heaven.

# Table of Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

Companies and universities replaced their aging mainframes with workstations and personal computers when the smaller machines proved to be cheaper, easier to upgrade, and more suited to individual users. Important advances that helped supplant the mainframes were fast networks and distributed software. Early distributed software packages included electronic mail, network file systems, and windowing systems.

Despite the proliferation of distributed computer systems, most programmers do not take advantage of their inherent parallelism and fault tolerance but continue to write only sequential, single-machine programs. Of course, not all programs can benefit from distribution, but we believe that most programmers view distributed systems as too complex for general-purpose programming. Indeed, the distributed programmer must worry about partial failures, asynchrony, concurrency, and, if working with groups of programming objects, consistency and ordering. We contend that many of these anxieties can be relieved by high-level language constucts that guarantee reliable, ordered data flow and a consistent system-wide membership.

## 1.1 Loosely Coupled Distributed Systems

A *distributed system* is a collection of independent computers that communicate only by shuttling messages around a network; in particular, the computers of a distributed system do not share a physical memory. In this thesis we consider systems with communications networks that are slow relative to their processor speeds, like modern workstations connected by Ethernet [MB76]. Such systems are called *loosely coupled*, and, because communication within them is slow, efficient applications must optimize network usage. Many distributed languages offer no assistance in this daunting task, and some actually work against the programmer by forcing inordinate and unnecessary communicaton.

For example, consider the *distributed shared memory* (DSM) abstraction [LH86]. In DSM, the memories of each computer in a distributed system are (conceptually) merged into a single memory, and programs access a memory cell without worrying about its physical location. The runtime system of a DSM language translates each read and write of a remote memory cell into a series of network transmissions. The DSM model is appealing because it blurs the distinction between local and remote memory and therefore makes a distributed system appear as a single, physically-distributed machine. In such a system, distributed programs appear sequential and can be written with familiar, well-known techniques. Unfortunately, message traffic in loosely coupled systems is inherently slow and, if a program written over DSM is at all efficient, its implementor has either detailed knowledge of the DSM runtime system or quite a bit of luck. Of course, programs which exploit details of a DSM runtime system bypass the primary benefit of DSM, *i.e.*, location transparency. Recently, researchers have realized the inherent inefficiency of DSM and have focussed on ways of relaxing memory semantics to improve performance [AHJ91]. These improvements reintroduce the disparity between distributed and stand-alone computers that DSM was designed to mask.

## 1.2 Distributed ML

The Distributed ML (DML) language is a general-purpose language whose primitives can be efficiently implemented on loosely coupled distributed computers. Programmers can use DML to write parallel, fault tolerant, and inherently distributed (*e.g.*, network management) code. DML adds one data object—a fault informative asynchronous multicast channel called a *port group*—to the Concurrent ML (CML) [Rep91,Rep92] language, an extension of the Standard ML (SML) [MTH89] general-purpose sequential language. In the rest of this introduction we discuss the four features that distingish DML from other distributed languages: asynchronous communication as a primitive, group communication as a primitive, support for consistent system-wide membership information, and CML (hence SML) as a language subset.

## 1.3 Asynchronous Communication

In many distributed languages, data senders (processes [Hoa78], statements [Inm84], or objects [Hut87]) block (delay) until a receiver has either started to react or formally replied. Such mechanisms enforce *synchronous communication*, and are popular because they have simple reliability and ordering semantics [Nel81,BN84]. In these systems, data senders need not worry that a hardware failure or buffer overflow will prevent information from reaching their receivers because they receive an explicit end-to-end acknowledgement that their communication has completed successfully [SRC84].

Unfortunately, it is difficult for programs that use synchronous primitives to exploit the full bandwidth of fast computer networks [TvR88,BvR92] because each communication involves one or more round-trip packet exchanges. Experiments by Thekkath and Levy [TL93] show that newer network technologies like FDDI [Ros86] and ATM [Min89] transfer large packets much quicker than Ethernet [MB76]; however, controller latency—the time used by I/O controllers to transfer data from memory to the network—has only improved slightly during the last decade. A distributed program cannot capitalize on higher bandwidth if every communication must incur a round-trip packet latency of the network. This overhead is even larger for languages that support selective communication [Kna92], which requires synchronization between, in general, more than two processes[1].

In fact, synchronization often needs to be achieved only at particular points in a stream of remote sends (*e.g.*, before dispensing cash from an automatic teller, before confirming a reservation to a ticketing agent, or before closing a file), and orderings enforced by synchronous primitives are overly restrictive. In DML programs, senders do not block, and explicit stability and ordering guarantees are used instead of the related properties inherent in synchronous communication. This allows the DML runtime system to pack messages together and use streaming techniques, improving the performance of DML programs. DML provides mechanisms that programmers use to mark the stability and ordering constraints necessary for program correctness.

## 1.4 Group Communication

Perhaps the most novel feature of DML is its support of one-to-many communication, or *multicast*, as a primitive operation. Only a few distributed programming languages that we know of have this feature [Geh84,LC84,Coo90], and none of them

[1]For instance, if $P_1$, $P_2$ and $P_3$ are communication endpoints, $P_1$ uses selective communication to specify that it wants to either send information to $P_2$ or receive information from $P_3$. The runtime system determines if $P_2$ and $P_3$ wishes to communicate with $P_1$, and commits $P_1$ to the conversation that it can initiate most quickly.

provide DML's elaborate multicast ordering support. However, many distributed programmers in different application domains use multicast in their systems, and we believe that their experience justifies DML's multicast support.

Indeed, programming experience has initiated and directed many language improvements since the 1950's. For instance, FORTRAN [Ett87] did not allow function recursion, but experience with this concise and intuitive programming technique has influenced modern language design. Similarly, early languages did not support multiple name spaces; large applications, written by many different programmers, contained name conflicts and these clashes had to be fixed manually. Modern languages, like Common Lisp [GLS90], Ada [Dep83], Modula-3 [Nel91], and SML [MTH89], provide name space grouping mechanisms that prohibit identifiers in one programmer's name space from conflicting with those in another.

Multicast communication is needed in a number of different distributed programming problems [Bir91]. For instance, data is often replicated among several machines to increase its locality, availability, or persistence; multicasting serialized updates to the data is a natural way to program such a system. Sometimes a set of machines is used to subdivide several computations, and the work must be distributed to prevent any single machine from becoming overloaded. Multicasting requests is a convenient way to start a computation on a set of server machines. In some distributed object-based settings, object components span machines, and multicasting requests to the components can be used to invoke an object's behaviour.

Of course, multicast constructs can be composed in any language that provides unicast primitives. However, there are at least two reasons that naïve implementations of multicast are inadequate: they allow messages to arrive in confusing orders and cannot exploit hardware multicast. Consider, for example, a set of machines $S$ that are maintaining a replicated copy of a variable $x$. If machines $M_1$ and $M_2$ respectively multicast "`x := 0`" and "`x := 1`" to $S$ using a series of unicasts, the updates may be received in different orders at members of $S$. After the multicasts complete, some machines might conclude that x is zero and some that it is one. The imposition of ordering properties makes efficient, asynchronous group communication useable, but they require protocols that are relatively complex, especially when the protocols must be fault-tolerant. In fact, multiple ordering protocols must be aware of each other to prevent destructive interactions [MPS92]. DML users are shielded from these complexities and can choose from a number of efficient and useful ordering primitives. In the preceding example, if $M_1$ and $M_2$ multicast their updates within a totally-ordered DML port group, their messages would be received in the same order at each member of $S$. Each machine would agree on x's value, which might be either zero or one, depending on the way the DML runtime system decided to order the updates.

Because it is almost impossible to reconstruct a multicast once it has been disassembled into unicasts—especially if communication is synchronous—languages that do not have a multicast primitive cannot use hardware multicast protocols [DC90] without forcing programmers to shun the language's primitives and use a special multicast library. Hardware multicast protocols send a multicast through the network as quickly as a unicast. Because DML allows programmers to use multicast explicitly, the language's runtime system can employ hardware multicast when appropriate.

## 1.5 System Membership Information

Most programming language definitions do not specify how implementions should behave if a system component (*e.g.*, a machine or a process) fails during execution; those that do usually require failures to be treated as exceptional events that are either masked or treated as fatal. We believe that this is the wrong approach: as the number of machines in a distributed system grows, the probability of experiencing a single failure increases. A language implementation that cannot tolerate failures is not scalable and therefore limited in its applicability—a shame, since distributed systems provide a manifest opportunity for high availablity and cheap persistence. Moreover, many distributed programmers do not want or need failures masked from their applications, so implementations that achieve transparent fault tolerance incur unnecessary costs and are of limited use.

Implementations of DML provide a system-wide *membership service* that supplies programs with a stream of dynamic membership information. DML programmers can use this stream to build fault-tolerant applications or can ignore it if they do not need it; libraries built on the DML primitives can provide transparent fault-tolerance. Because DML programs receive membership information from a single (fault-tolerant) membership service, all components of a program see the same set of failures and therefore need not explicitly coordinate before reacting to a membership change. This behaviour differs, for instance, from a program that uses time-outs to detect machine failures.

## 1.6 Concurrent ML and Standard ML

Distributed ML is an extension of Concurrent ML [Rep91,Rep92], a language designed to support node-level (lightweight) concurrent programming. CML itself is an extension of the Standard ML [MTH89,MT91] language, a general-purpose mostly functional language known for its polymorphic types and flexible module system. CML and DML are programming *languages* because they advocate and support a programming paradigm, that is, a way of reasoning about and developing concurrent and distributed programs, respectively. In contrast, Standard ML extended with Unix [RT74] system calls is not a distributed language, even though one could use the extensions to write a distributed program. Such a conglomeration provides neither sufficient type support (for SML, at least type safety and perhaps a form of polymorphism) nor the control structures (*e.g.*, expressive selective communication) of a distributed programming language like DML.

The most appealing features of SML to the distributed language designer is the existence of CML. Because a typical distributed program is inherently concurrent—it executes on multiple machines and communicates using arbitrary transmission patterns—any general distributed language must support node-level (lightweight) concurrency[2]. Concurrent ML provides this support to DML programmers. CML is implemented on SML/NJ [AM87,AM91], which uses a continuation-passing style compiler [App92], so a thread switch in CML is as fast as a function call. CML's first order synchronous events [Rep89] combine abstraction with synchrony through a concise yet powerful notation. DML extends CML's event notation and introduces prioritized *ordered events*: ordered events reduce the nondeterminism of CML's selective communication and preserve precedence between threads in a preemptive environment.

Although DML borrows many of its ideas about asynchronous group communication and system membership from the Isis Distributed Toolkit [BJ87,BCG91], DML programs are structured differently—they can be more abstract and hence easier to understand, maintain, and write—than Isis programs. DML's ordered events and preemptive threads allow distributed programs to employ CML's elegant programming style while exploiting modern, asynchronous group communication technology.

[2]One could imagine a system in which each node contained a single thread that polled an external buffering service for waiting requests and processed each individually. We would not consider such a system general purpose; it would prohibit synchronous communication—indeed, all potentially blocking code—and could not tolerate long-running programs. However, the X Window System [SG90] server (which is not intended for general purpose programming) is implemented in essentially this way.

## 1.7 Related Work

Many researchers have tried to integrate distributed programming into a high-level language. For instance, the Hermes project [ABG+91] shares many of the goals of DML. Hermes is completely type-safe, but some of its safety is attained at the expense of expressiveness and efficiency. In Hermes, distribution and fault-tolerance are automatic: the system (with human assistance from a "tuning expert") decides how to map numerous processes onto available machines, and uses automatic logging and checkpointing to cope with failures. In contrast, DML takes a more flexible, toolkit approach which avoids prejudging too many decisions for the programmer. It provides lower-level distribution and fault-tolerance mechanisms that can be combined using higher-order functions and events. We claim that programmers can build very high-level, general abstractions that apply the most appropriate and efficient distribution and fault-tolerance technqiues to the task at hand.

Wing, Nettles, and others [CLNW89,NW91] have investigated fault-tolerant distributed programming in Common Lisp and SML. Avalon/Common Lisp provides remote evaluation, clearly demonstrating its utility, but highlighting the semantic complexity arising when Common Lisp is used for distributed programming. Their SML-based work showcases the modularity benefits of SML by providing an atomic transaction system with replaceable components. This allows the programmer to tradeoff transaction properties for performance. Although transactions are suitable for a large class of applications, we believe that they restrict the available parallelism and overemphasize the update of mutable objects [Bir91].

Facile [GMP89,GMP90] is a concurrent functional language, sharing many of the goals of CML, but lacking events (see Section 3.2). Recently, Kramer and Cosquer [Kra92] built a prototype distributed implementation of Facile, which provides typed point-to-point channels. Their system permits most SML values to be transmitted, but provides limited support for fault tolerance. Communication in Facile is based on synchronous message passing and rendezvous, which, as these researchers themselves note [Kna92], causes significant costs compared with asynchronous communication, which DMLsupports.

Several earlier projects, including Argus [Lis88], SR [AO85], and Orca [BT88] have taken a language-based approach to fault tolerance and distributed programming. Many of these efforts are either based on languages, such as CLU and C++, that lack the power and generality of a functional, higher-order language such as SML, or do not benefit from ordered, asynchronous group communication.

Three of DML's distinguishing design principles—explicit failure notification, asynchronous communication, and group-based communication—are present in the Isis Distributed Toolkit [BJ87,BCG91]. Isis supports the virtual synchrony model of distributed communication, defined in Section 6.7, and it has been shown to be applicable to problems in many different application areas [BC90]. However, Isis is written in C and provides node-level (light-weight) concurrency with a nonpreemptive scheduler. Disturbingly, the correctness of many Isis programs depends on this lack of preemption; moreover, Isis will not function properly if its scheduler cannot swap tasks at regular intervals. Isis' group support extends only to Unix processes; finer grained objects cannot be first-class members of a group, and, because of this restriction, Toolkit applications are more difficult to prototype and test. We believe that, by supporting virtual synchrony and other communication models at the language level and by abstracting the Isis concepts, distributed programs can be written which are easily understood and provably correct.

The Horus project [vRBGR93], billed as Isis' successor, preserves the fundamental principles of its parent system—including explicit failure notification, asynchronous communication, and group-based communication—while removing its shortcomings, like Isis' dependence on nonpreemption. Although Horus and DML were designed concurrently, the two implementations are similar. Horus has focussed on implementation efficiency and modularity, while DML has centered on expressiveness and other application programming issues (the initial implementation of DML was completely slightly before that of Horus). The DML group communication model is more general because it permits communication patterns that are not virtually synchronous.

## 1.8 Thesis Contributions

This thesis discusses the design and implementation of a new distributed programming language called DML, used to program sets of workstations connected through a relatively low-speed network. It adds an asynchronous group-based communication object to the Concurrent ML language and provides ordered event objects for constructing abstract communication patterns. DML is the first programming language to provide an asynchronous, fault-informative, multicast object, and the use of ordered events is DML's unique and powerful way of encapsulating ordering information within application programs. DML's primitives are both powerful and general, and DML programmers can write simple programs to solve a wide variety of difficult distributed problems.

We introduce a formal notation for reasoning about the possible communication histories in DML programs and show that, using a very natural restriction of the DML syntax, the runtime system enforces the *virtual synchrony* communication model made popular by the Isis Distributed Toolkit [BJ87,BCG91]. We believe that ours is the first definition of virtual synchrony that captures its intuitive meaning; previous formal definitions are combinations of message existence and ordering constraints that fail to convey its purpose or usefulness.

Implementing a fault tolerant distributed system is difficult; resilient protocols are usually complex [NT88], and systems are difficult to modularize [HS93]. We describe how the implementation of DML's primitives is divided into layers and delineate what guarantees each module provides to the layer above it. We believe that the DML architecture is an excellent way to subdivide a system with many objects that communicate in complex ways. As part of its implementation, DML provides a *network simulator* so DML programs can be tested on a single machine before they are migrated to a distributed system.

# Chapter 2

# Standard ML

Although the programming innovations presented in this thesis are largely language-independent, we chose to develop and implement an extension to the Standard ML language [MTH89] rather than extend a typical system development language like C [KR88] or create a new programming language from scratch. Standard ML is a mostly functional language supporing strong static typing, polymorphism, updatable memory, exceptions, and an innovative module facility. SML evolved from the Meta Language of LCF [GMW79], a theorem proving system developed at Edinburgh. As mentioned in Section 1.6, SML is a good choice because of the substantial unsatisfied demand for distributed programming among SML users and the existence of CML [Rep92], an efficient, high-level thread package implemented in SML. Additionally, SML's type system, higher-order functions, module system, and efficient implementation make it easier to prototype and subdivide implementation tasks.

In this chapter, we introduce the SML syntax necessary to understand the code examples in the rest of the thesis. A thorough introduction to SML can be found in Paulson's book [Pau91], and the language is formally defined in [MTH89] and [MT91].

## 2.1 Base Values and Types

As mentioned above, Standard ML is a mostly functional language: many SML programmers do not track the state of memory and instead write *expressions* that, without producing any side effects, compute *values*. SML does provide some imperative features—it would be impossible to explicitly express communication between two independent processes without state—and we will examine such exceptions in Section 2.6.

All control constructs in SML are expressions, and each has an evaluation rule to generate a result of a specific *type*; sometimes we call this the type of the expression itself. For instance, the code

```
if test then code1 else code2
```

is an `if-then-else` conditional expression. The expression is evaluated by first evaluating the boolean-typed `test` and, based on the result, evaluating the appropriate branch. The type of the conditional is the type of both `code1` and `code2`: both branches must have the same type.

SML has a standard collection of base (ground) types and associated constants, including integers (type `int`), floating point numbers (`real`), strings (`string`), and booleans (`bool`). The least understood SML base type is `unit`; only one value, written as `()`, has type `unit`. Functions that are used exclusively for side effects, *e.g.*, the `print` function, return the unit value.

The `val` declaration is used to associate names with values. Declarations usually appear within a `let` expression:

```
let
  val test = true;
in
  if test then 1 else 0
end
```

The type of a `let` expression is the type of its body, that is, the expression between the `in` and `end` keywords. This `let` expression binds the value `true` to the name `test`:

this is sometimes called *single assignment* because, unlike assignment in languages like C, `test` does not refer to a memory location and therefore its contents cannot be changed. SML has a different mechanism—indeed, a special type—for handling updatable memory cells.

Sometimes we will want to indicate the value and type of a named value without enclosing it in a `let` expression. The notation

```
val test = true: bool
```

indicates that `test` has the value `true` and is a Boolean. This syntax is also used to explicitly assign types to SML values. In our examples, we have not had to declare types at all—unlike most other typed languages, including C—because SML infers types automatically (we will mention this again in Section 2.3). However, types can be explicitly declared, and such declarations often serve as useful documentation. We can also indicate the type of a named value without writing it out, which is useful if the value is complex, like a function. The syntax

```
val test: bool
```

is called a *type specification* and indicates that the value bound by `test` has type `bool`. Type specifications have a number of uses; as we will see in Section 2.8, they are the way code modules' interfaces are delineated.

## 2.2 Composite Types

SML supplies several *composite* types, *i.e.*, types composed of multiple, less complex, types. For instance, the value `(true, "maroon")` has a *product type*, `bool * string`. Product types can have any number of components, but for large composite types, the product type juxtaposition notation becomes inconvenient and confusing. In SML, *record types* provide names for their components, called *fields*: the value

```
{test = true, color = "maroon"}
```

is a record and has type {`test: bool, color: string`}. If the value above were bound to the name `x`, the notation `#test x` could be used to extract the value of the `test` field—*i.e.*, `true`—from the record `x`.

Function types are also composite types; in SML, functions can be declared with the keyword `fun`. A function that computes the average of two integers could be coded as

```
fun avg (x, y) = (x + y) div 2;
```

Note that this function takes a pair of integers and returns an integer; its type is written as `int * int -> int`. The `fun` keyword is shorthand, and the definition above is equivalent to

```
val avg = fn (x, y) => (x + y) div 2;
```

This latter notation is similar to the $\lambda$-calculus [Bar84], where the keyword `fn` is $\lambda$. Both versions of `avg` have the specification

```
val avg: int * int -> int
```

As with most modern programming languages, SML functions can call themselves, and recursion is the way most SML functions loop. The function

```
fun fact x = if x = 0 then 1 else x * (fact (x - 1))
```

computes the factorial of positive numbers and diverges for negative numbers. Its specification is

```
val fact: int -> int
```

SML provides a special notation for defining functions that return other functions. Consider, for example:

```
fun f x = fn y => x + y + 1;
```

This function takes an argument $x$ and returns a function that returns its argument plus $x + 1$. It's specification is

```
val f: int -> (int -> int)
```

We could have alternatively defined `f` by

```
fun f x y = x + y + 1;
```

A function that returns another function is called *curried.* Although the latter syntax makes `f` appear as if it takes two arguments, it does not; both it and its return value accept only one argument (in fact, all SML functions take one argument and return one result). The concept of currying originated in the lambda calculus [Bar84].

## 2.3 Polymorphism

Unlike many languages, SML does not require the types of names and variables to be declared before they are used[1]. Instead, the SML compiler *infers* types. In the factorial example above, the type of `x` had to be `int`; it was being compared to the integer zero, and the integer one was being subtracted from it. What would have happened, though, if we hadn't provided enough information for the compiler to infer the type? Consider the function

```
fun identity x = x;
```

This function would work properly regardless of its argument, and therefore the compiler cannot infer a specific type for it. Instead, SML assigns `identity` the *polymorphic type* $(\forall\alpha)(\alpha \rightarrow \alpha)$, which is specified as

```
val identity: 'a -> 'a
```

The SML compiler automatically infers the most general type possible for a given value. Functions with polymorphic parameters can be used with differently-typed arguments within the same code block:

```
let
  val identity x = x;
in
  identity true;
  identity "maroon"
end
```

[1]Unless overloaded functions are used; see [Pau91] for more details.

As another example of polymorphism, consider the divergent function

```
fun f x = f (x + 1);
```

The compiler knows that x must be an int, but cannot determine a specific type for f's return value. This function has specification

```
val f: int -> 'a
```

Because f never terminates, it is safe to use its return value in any context, *e.g.*, as the argument to any function.

## 2.4 Datatypes

Tuples and records define product types; each tuple and record value contains multiple fields, one field for each component type. In SML, *datatypes* also consist of component types, but datatype values contain only one field. Datatypes are similar to variant records in Pascal [FW85] but are tagged so that the type system cannot be bypassed. Consider, for instance the type of wishes:

```
datatype wish = PEACE
              | MONEY of real
              | SLEEP of int;
```

This definition states that people either wish for peace, for some (real) amount of money, or to sleep for some (integer) length of time. The identifiers PEACE, MONEY, and SLEEP are called constructors and serve a dual purpose. They can be used as functions to construct wishes—MONEY, for instance, has type real -> wish—and they are used in pattern matching, as we will see in Section 2.5. Note that a single wish cannot be for peace, money, and sleep; in particular, PEACE is a wish (*i.e.*, it has type wish) all by itself.

Datatypes can be used to define recursive types. Suppose, for instance, that we were not content to choose between wishing for peace, money, and sleep, and that we also wanted to wish for more wishes. We could redefine type wish as:

```
datatype wish = PEACE
              | MONEY of real
              | SLEEP of int
              | WISH of wish * wish;
```

Now we can request both peace and and a good night's sleep with only one wish:

```
val w = WISH (PEACE, SLEEP 8): wish
```

Recursive datatypes are used to define many useful data structures, like lists, trees, and graphs.

### 2.4.1 Predefined Datatypes

SML has a number of predefined datatypes. The type `bool` is not really a base type; actually, `bool` is a datatype:

```
datatype bool = true | false;
```

Datatypes can be *parameterized* by polymorphic type variables. For instance, the `option` datatype is predefined as

```
datatype 'a option = NONE | SOME of 'a
```

Programmers often use the `option` datatype to code functions with optional parameters. For instance, an `int option` parameter can either be specified with `SOME x` for some integer $x$ or left unspecified with `NONE`. Note that the values `SOME true` and `SOME "maroon"` have *different* types; the former is a `bool option` and the latter is a `string option`.

Perhaps the most important predefined datatype is `list`, defined as

```
datatype 'a list = nil | :: of ('a * 'a list)
infix :: 5
```

An SML list is either the constructor `nil` or

```
x1 :: x2 :: ... :: xn :: nil
```

for a series of same-typed expressions $x_1, x_2, \ldots, x_n$. Note that `::` is declared to be *infix*, *i.e.*, it is placed between its arguments rather than in front of them. The `::` operator associates to the right. Lists are so important in that they have been allocated special SML syntax: the text `[]` is the same as `nil`, and

```
[x1, x2, ..., xn]
```

is shorthand for

```
x1 :: x2 :: ... :: xn :: nil
```

## 2.5 Pattern Matching

SML provides a powerful mechanism called *pattern matching* to decompose complex values and bind their components to symbolic names. Pattern matching is most commonly used when defining functions. For instance, we can define a function on wishes like this:

```
fun grant PEACE = "Done.  What will you humans do now?\n"
  | grant (MONEY m) = "Alakazam; here's $" ^ (makestring m) ^ "\n"
  | grant (SLEEP _) = "Rock a bye, baby...\n"
  | grant (WISH (w1, w2)) = (grant w1) ^ (grant w2);
```

The `grant` function has type `wish -> string`. Note that we are really defining four separate `grant` functions, one for each type of wish, and the pattern matching syntax of `grant` is more concise and convenient than a single function definition with an imbedded case or switch statement, although SML does provide a `case` statement for explicit pattern matching within a function. Pattern matching is particularly useful with functions defined on lists:

```
fun length [] = 0
  | length (x :: l) = length l + 1;
```

In fact, `length` and a number of other list utilities are predefined in SML:

```
val map: ('a -> 'b) -> 'a list -> 'b list
val app: ('a -> 'b) -> 'a list -> unit
val rev: 'a list -> 'a list
```

The curried `map` and `app` routines apply a function to each element of a list; the former returns a list of the results while the latter discards them. The function `rev` reverses a list.

Pattern matching can be used to disassemble any composite type. The first function we defined in Section 2.2 used pattern matching:

```
fun avg (x, y) = (x + y) div 2;
```

Here, avg has one argument—a pair—and pattern matching is used to name each component of the pair. In this sense, all SML functions takes only one argument; tuples and pattern matching can simulate multi-argument functions (as can curried functions, see Section 2.2).

Records can be decomposed with pattern matching, too:

```
fun f {test = t, color = c} = if t then c else "";
```

SML's pattern matcher disassembles f's record argument, binding the name t to the Boolean value of the field test and c to the string value of the field color. Some shorthand makes records easier to disassemble and use:

```
datatype t = T of {test: bool, color: string}
fun f (T {test = test, color = color}) = test;
fun g (T {test, ...}) = test;
```

Functions f and g are the same. In the definition of g, SML substitutes test=test for test and expands the ellipsis to all field names that are not declared explicity—in this case, to color = color. Note that the type of g's argument must be completely specified before SML compiles it. That is, if the T constructor had not been used in this example, SML would have complained that it couldn't determine the type of g's argument: there are an infinite number of record types with a (boolean) test field. Datatypes are often used to wrap records so that functions defined on the records can use the ellipsis syntax.

Pattern matching can also distingish constants:

```
fun english 0 = "zero"
  | english 1 = "one"
  | english _ = "something other than zero or one";
```

In SML, the underscore character is a wildcard and matches anything not matched by a previous definition.

## 2.6 Imperative Features

The Standard ML language is not completely functional; in this chapter we discuss several functions that cause side effects. Because imperative features are necessary to print to the screen and share data between running processes, a purely functional language can be used for neither interactive nor general-purpose distributed programming[2].

Perhaps the most used imperative functions in SML are the output functions, like `print`, that write data to files and to users' screens. SML also provides updatable memory cells called *references*, groups of references called *arrays*, and *exceptions* for redirecting control and passing along information about unexpected errors. The SML/NJ implementation also provides flow control constructs called *continuations* that CML uses to implement lightweight threads. Imperative data types are not fully polymorphic in SML; instead, the language introduces a restricted form of polymorphism for these types.

### 2.6.1 References and Weak Types

The expression `ref 1` is a memory cell with the integer one stored in it, and it has type `int ref`. References can be dereferenced with the `!` function, and their contents can be altered with `:=`. As an example, the value of the expression

```
let
  val cell = ref 1;
in
  cell := 2; ! cell
end
```

is two. The `ref` function is also a constructor and can be used in pattern matches; it and the other reference operators have the following type specifications:

[2]It *is* possible to do explicit parallel programming in a functional language; for example, Par-Alfl [Hud86] programmers can assign subexpressions of a functional program to different processors and have each evaluated simultaneously. However, parallel threads cannot communicate during execution: this would cause side effects.

```
val ref: '1a -> '1a ref
val ! : 'a ref -> 'a
val := : 'a ref * 'a -> unit
infix :=
```

The polymorphic type variable in the specification of `ref` is called a *weak* type variable, and it deserves some attention since weak types permeate CML and DML code.

Weak type variables are unique to the SML/NJ implementation[3] and are a generalization of *imperative* type variables [Tof90] present in SML's definition [MTH89] and all other implementations. To see why `ref` cannot have the type `'a -> 'a ref`, consider:

```
let
  val cell = ref (fn x => x);
in
  cell := (fn y => 0);
  ((! cell) (fn z => z + 1)) 0
end;
```

Observe the operation and type system behaviour during the evaluation of this expression:

1. The name `cell` is bound to a memory location containing the identity function (recall that `fn x => x` is $\lambda x.x$). If `ref` has type `'a -> 'a ref`, `cell` has type `('a -> 'a) ref`. At this point, `cell` is capable of holding *any function of type* `t -> t` *for any* `t`. In particular, the function it contains need not have polymorphic type; it could have type `int -> int`.

2. The `cell` location is updated with the function that always returns zero. The type system notices that this function could be specialized to have type `int -> int` and so, as discussed above, does not object to the update.

3. The cell location is dereferenced, yielding a value of type `'a -> 'a` (because `!` has type `'b ref -> 'b`), and it is applied to the increment function. This

[3]Weak types were just recently proven safe [HMV93].

yields a value of zero, but the type system associates `int -> int` with it. As any C programmer knows, a function of value zero leads to trouble, and the final application of zero to zero is meaningless (and will probably cause a core dump).

The problem is that the polymorphism of `cell` is too general: it allows `cell` to be assigned a non-polymorphic function and then considers its contents to be fully polymorphic. Weak and imperative types prevent polymorphic memory cells from ever coming into existence, and SML's type system would not allow `cell` to exist with type `('a -> 'a) ref`.

Unfortunately, weak types have a way of percolating through interfaces. For instance, a function implemented with references may inherit weak types even if the function could be used safely in a fully polymorphic way (the function may even have a less-efficient but fully polymorphic implementation). In his thesis, Reppy had to prove that several of the CML polymorphic functions are safe even though he implemented them with unsafe fully polymorphic continuations [Rep92]. Wright has recently shown that eliminating weak and imperative types but restricting polymorphism to values alleviates the percolation problem [Wri93b]. With Wright's system, `cell` in the example above could not be assigned the type `('a -> 'a) ref` because it is the result of an *application*, *i.e.*, the return value of a function. Although this rule seems restrictive, Wright shows that his principle still type checks most SML packages, including CML[4].

## 2.7 Exceptions and Continuations

When an error or other unusual event occurs in an SML program, exceptions are used to reroute control and pass information about exceptional incidents. The `raise` con-

[4] Assuming that Wright proves his type system safe, which he claims is "easy" [Wri93a], the fact that it type checks CML makes Reppy's safety proof unnecessary.

struct redirects control from the point where an error is detected to the appropriate exception handler, defined with `handle`. For example:

```
exception Invert
fun invert 0.0 = raise Invert
  | invert x = 1.0 / x;
```

The `exception` keyword declares a new exception named `Invert`, and the `invert` function will raise `Invert` if applied to the real number zero.

```
fun invertList [] = []
  | invertList (x :: l) = (invert x :: (invertList l))
      handle Invert => invertList l;
```

This function inverts a list of real numbers by applying the `invert` function to each element of the list. If one of the inversions causes the `Invert` exception to be raised, it is removed from the list.

Exceptions can carry values from a `raise` to a `handle` and can therefore be assigned types. In DML, the exception

```
exception DML of string
```

is used to signal events that should not occur and which indicate a bug in DML's runtime system. The `string` argument to the `DML` exception is a description of the problem. Exception arguments are extracted with SML's pattern matching facility. For instance:

```
(some_dml_code) handle (DML s) => print s
                      | x => ();
```

If a `DML` exception is raised during the execution of `some_dml_code`, the exception's argument is printed. If any other exception is raised, unit is returned and the exception is masked (in practice, it is not necessary to handle the `DML` exception as the CML runtime system does this automatically). The values associated with exceptions cannot be fully polymorphic without losing type safety; in SML, they are restricted to weak types. This thesis and the implementation of DML that it describes only use monomorphic exceptions.

SML/NJ provides one other imperative construct, the first-class continuation. Continuations are functions that encapsulate a partially evaluated computation, and they are used heavily in the SML/NJ compiler within intermediate code for optimization and code generation [App92]. Although we do not use them in this thesis or directly in DML's implementation, we mention them because they make CML easier to implement efficiently [Rep92].

## 2.8 Structures and Signatures

Standard ML programmers divide their large programs into manageable chunks using the language's module system. Structures, signatures, and functors constitute a calculus of name spaces; these name spaces are manipulated at run time, and the system is expressive enough so that name spaces can be combined, shared, and presented in many different useful ways. The modules system also provides a separate compilation facility and is useful for managing applications that are written by multiple programmers.

A *structure* is a set of bindings that constitute a single programming unit and contains mappings from names to exceptions, functions, values, and even other structures. For instance, consider the structure:

```
structure Crowd =
  struct
    exception NoVictim

    datatype person = female | male;
    type crowd = {females: int, males: int};

    fun make () = {females = 0, males = 0};
    fun add ({females, males}, female) =
        {females = females + 1, males = males}
      | add ({females, males}, male) =
        {females, males + 1};
    fun shoot ({females, males}, female) =
```

```
        if females = 0 then raise NoVictim
        else {females = females - 1, males = males}
      | shoot ({females, males}, male) =
        if males = 0 then raise NoVictim
        else {females = females, males = males - 1};
  end;
```

The structure `Crowd` contains mappings for the exception `NoVictim`, the types `person` and `crowd`, and the functions `add` and `shoot`. *Signatures* are interfaces to structures; they describe structures like types describe values. For instance, the signature `CROWD` below describes the `Crowd` structure.

```
signature CROWD =
  sig
    exception NoVictim
    type person
    type crowd

    val male: person
    val female: person

    val make: unit -> crowd
    val add: crowd * person -> crowd
    val shoot: crowd * person -> crowd
  end;
```

In fact, the `CROWD` signature describes any structure that provides the exceptions, types, values, and substructures declared in its body. Thus, code that uses a structure of signature `CROWD` can be written without knowing its implementation, and `Crowd` can be reimplemented without causing type errors in programs that rely on it. Because structures and signatures can encapsulate multiple types, values, exceptions, and structures, they are more general than the abstract types of other languages.

A *functor* is a parameterized structure, that is, a function from structures to other structures. Many program units depend on other units for functionality, and functors conveniently express these dependencies while maintaining the independence between structure implementations. Because we do not use functors in the text of this thesis, we will omit an extensive example; see [Pau91] for more information.

# Chapter 3

# Concurrent ML

Because distributed systems consist of multiple computers that run simultaneously, they are inherently concurrent. Any general purpose distributed language must be designed under the assumption that concurrency will disseminate and can therefore exist on a single machine. For instance, if two machines simultaneously send a message to a third machine, the latter inherits concurrency from the system and must have some method—*e.g.*, nondeterministic or unreliable delivery semantics, message queues, or lightweight threads—for dealing with simultaneous communication partners.

Distributed ML is an extension of Concurrent ML (CML), which in turn is an extension of Standard ML, discussed in Chapter 2. CML is a language designed for writing programs that are naturally concurrent, like windowing systems and other interactive systems [Rep92]. All CML programming constructs are available to the DML programmer, and much of the DML interface is defined in terms of CML objects, such as *events* [Rep89]. CML runs on a single computer, and CML programs simulate concurrency by creating light-weight threads to evaluate functions in parallel. Threads converse through typed channels, and all communication in CML is synchronous, like that of CSP [Hoa78].

As was done in Chapter 2 with SML, this chapter introduces the CML constructs and syntax necessary to understand the examples in the rest of this thesis. A more thorough discussion of CML appears in the CML manual [Rep90] and Reppy's thesis [Rep92].

## 3.1 Threads and Channels

An SML program computes by evaluating a single expression. In contrast, a CML program computes by concurrently evaluating multiple expresions. Such expressions may not actually be evaluated in parallel—the original CML implementation runs on a single uniprocessor—but every CML runtime system must at least simulate parallelism and make concurrent expressions appear as if they are evaluated simultaneously. In particular, the language specification forces uniprocessor CML implementations to provide a preemptive scheduler so that, *e.g.*, a single long-running expression evaluation cannot prevent other expression from being executed.

Expression evaluators are called *threads* and are created with the `doit` and `spawn` functions.

```
type thread_id

val doit: (unit -> unit) * int option -> unit
val spawn: (unit -> unit) -> thread_id
```

The `doit` command creates a CML program's initial thread, which executes by calling `doit`'s first argument (the second argument is the time between scheduler preemptions and is not important to this discussion); `doit` does not terminate until the CML computation is complete[1]. A thread can create a child thread by calling `spawn`: the new thread applies `spawn`'s argument to unit and executes concurrently with its parent. Unlike `doit`, `spawn` never blocks. The `thread_id` returned by `spawn` can

[1]The `doit` routine does other administrative tasks besides creating the initial CML thread and is only called once, to start a CML computation.

```
let
  fun sender () = let
    val ch = channel ();
    fun accepter () = accept ch
  in
    spawn accepter;
    send (ch, ())
  end
in
  doit (sender, SOME 20)
end;
```

Figure 3.1: Communicating threads

be used to compare threads and to detect thread termination (*i.e.*, the successful evaluation of `spawn`'s argument).

Spawned threads communicate with each other by sending messages through typed *channels*. Channels are created by the `channel` routine, and data is exchanged through them with `send` and `accept`.

```
type 'a chan

val channel: unit -> '1a chan
val send: ('a chan * 'a) -> unit
val accept: 'a chan -> 'a
```

Both `send` and `accept` can block. That is, a thread $t$ cannot communicate without a partner; if $t$ attempts to send a message on a channel and no other thread is waiting to accept its message, $t$ blocks until it can communicate. Symmetrically, a thread $t$ attempting to accept a message from a channel blocks until one becomes available. Note that channels created by `channel` have a weak polymorphic type (see Section 2.6).

Consider the CML program in Figure 3.1. The function `sender` declares a channel `ch` and a function `accepter`, where `accepter` accepts the next value on `ch`. It then spawns a new thread to evaluate `accepter` and attempts to send unit on `ch`, forcing a synchronization with its child thread. The `doit` routine is used to initiate the computation by starting the `sender` thread.

## 3.2 Events

Blocking send and accept operations are useful when a program's communication patterns are known and fixed at compile time. However, some programs, especially those that deal with external inputs like users and clocks, must use *selective communication* to work efficiently and properly. Selective communication allows a thread to accept the first message available on a set of channels or send a message only if another thread is willing to accept it within five seconds.

Unlike most concurrent languages, CML provides a general selective communication statement by introducing objects that represent *potential blocking actions.* Such objects have type `event` and include the potential to send or receive on a channel, wait for a timeout, or detect that a thread has exited.

```
type 'a event
type time

val transmit: ('a chan * 'a) -> unit event
val receive: 'a chan -> 'a event
val timeout: time -> unit event
val threadWait: thread_id -> unit event

val select: 'a event list -> 'a
val sync: 'a event -> 'a
val poll: 'a event -> 'a option
```

If a blocking operation returns a value of type `'a`, then the potential to commit to that operation has type `'a event`. For instance, we saw in Section 3.1 that the `send` command returns unit; the return value of the `transmit` routine, which represents the *potential* (or capability) to send, has type `unit event`. Similarly, the `receive` command returns an event that is the potential to accept a value from a channel.

The `timeout` function returns a potential to continue after a certain length of time, and the `threadWait` routine returns a potential of waiting for a thread to terminate.

The `transmit` and `receive` functions do not actually initiate communication—they merely package up and return the ability to converse on the specified channels. The `select` routine is used to realize an `event`'s potential and invoke CML's general selective communication mechanism. Presented with a list of `events`, `select` determines the first of them that is ready for synchronization and commits its operation, returning the result. If more than one of its arguments are ready simultaneously, it nondeterministically chooses between them (which is inconvenient in some situations, see Section 4.6). The `sync` function is equivalent to `select` of a singleton list.

Events can be used to represent streams of data. For instance, the event value of `receive ch` remains the potential to accept a value from `ch` after it has been used. Thus

```
let
  val evt = receive ch;
in
  [sync evt, sync evt, sync evt]
end
```

returns a list of the next three values sent on `ch`. A `receive` event is a handle on all future values on its channel and therefore resembles a stream of data.

CML provides a pair of event combinators that promote the use of events in application-level modules and interfaces.

```
val choose: 'a event list -> 'a event
val wrap: 'a event * ('a -> 'b) -> 'b event
```

The `choose` function is to `select` as `transmit` is to `send`: `choose`'s return value event is the ability to apply `select` to its argument list. A call to `wrap` associates a post-synchronization function to an event; the result of

```
select [wrap (f, x)]
```

is the same as that of

```
f (select [x])
```

The `choose` and `wrap` operators allow CML programmers to build event-valued abstractions that behave as the base events provided by the language. For instance, consider the function

```
fun my_receive (ch, def) =
  choose [receive ch,
          wrap (timeout five_seconds, fn () => def)];
```

which has specification

```
val my_receive: 'a chan * 'a -> 'a event
```

When a thread commits to the return value of `my_receive`, it tries to accept from the channel `ch` but, if a communication partner isn't found within five seconds, the thread procedes with the default value `def`. Because the return value is an `event`, it can be used in calls to `select`, `sync`, `choose`, and `wrap` just as the events returned by the predefined event-valued functions. The ability to create communication abstractions with the same rights and abilities as the system defined ones is an important asset to CML programmers.

## 3.3 Active Objects

Many CML programs, including most of DML's implementation, is structured into *active objects*. Each object consists of a small amount of state and a thread of control which is the only reader and writer of the state. Other threads communicate with the object through a set of channels monitored by the object's thread. Consider, for example, the code to create and communicate with an active `stack` object presented in Figure 3.2. The thread implementing the stack, which repeatedly evaluates the `object` function, is spawned by `stack`. The stack's state is encoded in the argument to `object` and changes as the stack's thread sends and receives stack elements

```
datatype 'a stack =
  STACK of {inCh: 'a chan, outCh: 'a chan};

fun stack () = let
  val inCh = channel () and outCh = channel ();
  fun object [] = object [accept inCh]
    | object (x :: l) =
      select [wrap (receive inCh,
                    fn y => object (y :: x :: l)),
              wrap (transmit (outCh, x),
                    fn () => object l)];
in
  spawn (fn () => object []);
  STACK {inCh = inCh, outCh = outCh}
end;

fun push (STACK {inCh, ...}, x) = send (inCh, x);
fun pop (STACK {outCh, ...}) = receive outCh;
```

Figure 3.2: Active object implemention of a stack

through outCh and inCh, respectively. When the stack is empty (*i.e.*, when object's argument is null), the stack thread ignores outCh; otherwise, it uses select to watch for communications on both inCh and outCh. The push and pop functions (*methods*, in object-oriented terminology) hide the channels used to communicate with the stack object. Notice that push returns unit because it cannot block; pop, however, returns an event that can be used in selective communication.

# Chapter 4

# Distributed ML

In this chapter we introduce the concepts and syntax used by DML programmers to write distributed applications. We discuss *ports* and several ways of classifying them, and we examine how ports are created and used in a Distributed ML environment. In Chapter 5, we look at some real examples of DML code.

## 4.1 Ports

In any multithreaded system, individual threads must *communicate* and *synchronize* with each other to participate in shared computations. In DML, threads communicate through *ports* and synchronize by blocking on *ordered events*; we will study the latter in Section 4.6.

DML programmers create three types of ports: *source ports*, *destination ports*, and *meta ports.* Threads can share a piece of data by placing it on a source port and receiving it from a dest port. Meta ports are used to receive *meta information*, *i.e.*, data that reflects the creation or destruction of other ports in the system. We call the set of all ports in the system *port space.* Port space is dynamic, and ports can be created and destroyed at run time.

Table 4.1: Port space partitions

| **If a port belongs to...** | **then it...** |
|---|---|
| the set of source ports | is used to send data |
| the set of destination ports | is used to receive data |
| the set of meta ports | is used to receive data about other ports |
| port group $G$ | is used to communicate with other members of $G$ |
| ordering preserve $O$ | orders data relative to communication within $O$ |
| port colony $C$ | fails (and participates in thread-based ordering chains, see Section 4.4) with members of $C$ |
| heap $H$ | shares an address space (in Unix, a process) with members of $H$ |

Table 4.1 summarizes five important ways of partitioning port space, including the classification of ports by their ability to send and receive data. Port space can be partitioned into multicast channels called *port groups*, the units of communication sharing in DML. Data sent on a source port of a port group $G$ is *multicast* and will eventually be available at all extant destination ports of $G$. Meta ports record creation or destruction events that affect their port group.

As we will see in Section 4.4, DML provides powerful data ordering primitives to restrict the number of possible event interleavings in an asynchronous system. Each port group is assigned exactly one ordering, and the ordering properties are enforced across *ordering preserves*[1]. (*e.g.*, two totally-ordered port groups of the same preserve are totally-ordered with respect to each other). In general, each port group belongs to exactly one ordering preserve.

Port space can also be partitioned into *port colonies*, the units of failure in DML.

[1]The noun "preserve" is used here as in "wildlife preserve." It is not intended to suggest either the verb or "fruit preserves."

All ports used by a single unit of computation (typically a Unix process, but potentially groups of threads within a single process) share a port colony. Because port space is a global construct and can span separate computers, some ports may disappear when a machine crashes, a network link fails, or a program faults. Port colonies are used to bracket a set of ports whose members cannot exist in isolation, for example, ports used by the same thread in the same computation. If any element of a port colony $C$ fails, then every element of $C$ fails, and this knowledge is used to write shorter and more efficient programs. As we will see in Section 4.4, port colonies can also be used to limit causal ordering chains.

When a DML session is started on a single machine, the operating system allocates a portion of virtual memory—called a *heap*—and a single thread of control (in Unix, a heap is essentially a process). The thread of control can then execute a DML program by creating ports and port colonies within the heap, spawning threads that communicate through ports with other DML sessions, etc. Heaps can fail in several different ways; they can fail to catch a raised exception, run out of virtual memory, or be terminated by users (in Unix, with the `kill(1)` command). In general, heaps fail independently, and a port colony cannot span heaps[2]. DML provides a routine for simulating the failure of a single port colony without destroying the colonies that share its heap, so distributed and fault-tolerant DML programs can be developed within a single heap and then dispersed to other heaps after they have been tested.

As summarized in Figure 4.1, a port has several characteristics that distinguish it from other ports in port space. Each source, destination, and meta port belongs to a unique port group and a unique port colony; a port's ordering preserve and its heap are determined by its port group and port colony, respectively.

[2]Although one can imagine an operating system that forces "coupled" heaps to fail simultaneously, DML does not provide a syntax for declaring colonies that span sessions.

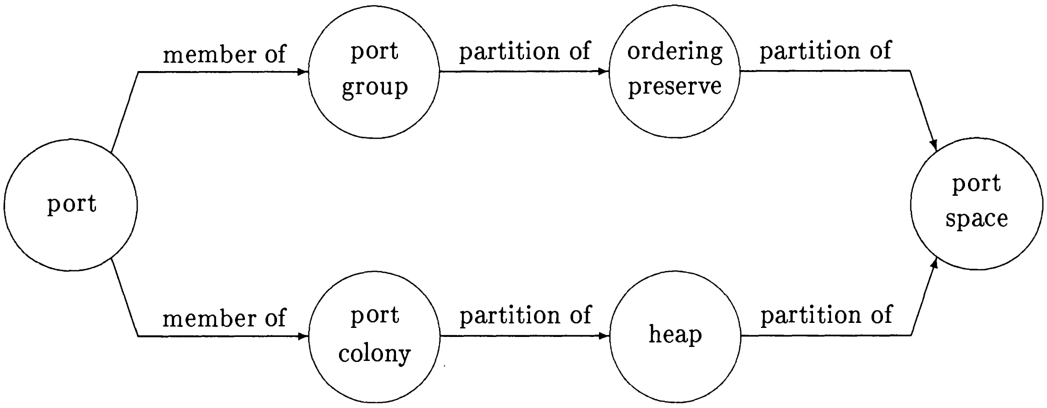



Figure 4.1: Classifying Ports and Partitioning Port Space

## 4.2 Declaring Ports

In this section we discuss those portions of the `DISTRIBUTED_ML` signature that allow programmers to declare source, destination, and meta ports. The relevant types and functions are listed in Figure 4.2.

Although port groups are multicast channels and CML has the channel type `'a chan`, no `'a port_group` type exists in DML; instead, a port group is described by its endpoints—values of type `src_port` or `dest_port`—and by groupviews of type `gview`. Groupviews are snapshots of a port group's membership and are useful because port groups dynamically add and remove member ports. DML provides functions (not shown in Figure 4.2) that return member lists relative to a specified `gview` value.

Port colonies have type `port_clny` and can be dynamically created with the `mkClny` function. Ordering preserves have type `ord_preserve`, and a port group's ordering preserve can be recovered by applying the `preserveOf` function to one of its groupviews. A port's heap is determined by the heap (in Unix, its process) on which it was declared; ports cannot migrate between address spaces.

Each port group has a group ordering of type `grp_ordering`, discussed in Section 4.4, that restricts the delivery order of data sent through the group. One impor-

```
signature DISTRIBUTED_ML = sig

  type gview

  type port_clny
  val mkClny: unit -> port_clny
  type ord_preserve
  val preserveOf: gview -> ord_preserve

  datatype ordering = FIFO | CAUSAL | TOTAL | UNIVERSAL
  type grp_ordering
  val vsync: ordering -> grp_ordering

  type port_context
  val NO_CONTEXT: grp_ordering -> port_context
  val PRESERVE: ord_preserve * grp_ordering -> port_context
  val VIEW: gview -> port_context

  type 'a grp_type
  val dml_string: string grp_type
  type 'a src_port
  type 'a dest_port

  exception StaleView
  exception WrongType
  val mkSrc: port_clny * port_context * '1a grp_type -> '1a src_port * gview
  val mkDest: port_clny * port_context * '1a grp_type -> '1a dest_port * gview
  val rmSrc: 'a src_port -> unit
  val rmDest: 'a dest_port -> unit

  type port_id
  val srcID: 'a src_port -> port_id
  val destID: 'a dest_port -> port_id
  val sameID: port_id -> port_id -> bool
  datatype meta_event =
    CREATE of port_id
  | REMOVE of port_id
  | FAIL of port_id;
  val mkMeta: port_clny * '1a gview -> (meta_event * '1a gview) dest_port

  ...
end
```

Figure 4.2: Most of the DISTRIBUTED_ML signature

tant group ordering, called *virtual synchrony*, is provided by the operator `vsync`, but others can be defined. The `vsync` procedure takes an `ordering` value, either `FIFO`, `CAUSAL`, `TOTAL`, or `UNIVERSAL`, which describes the normal (non-meta) data ordering within the group.

New source and destination ports must be created relative to others, if any, that already exist; new ports may share port groups, port colonies, or ordering preserves with predecessors. A *port context*, *i.e.*, a value of type `port_context`, describes the environment in which a new port is to be generated. The `NEW_CONTEXT` function takes a group ordering and returns an empty port context—ports created relative to it will belong to a new port group and a new ordering preserve. The `PRESERVE` function accepts an ordering preserve $O$ and a group ordering and returns a context $C$ that remembers $O$. Ports created relative to $C$ will belong to a new port group, and that port group's preserve will be $O$[3]. When given a groupview of a port group $G$, the `VIEW` routine returns a context that is used to create new members of $G$ (which inherit $G$'s group ordering and ordering preserve).

SML's type system is sufficiently strict that type information need not be available at run time; indeed, such knowledge is not available in most implementations of the language. Any function that manipulates data assumes that the data's type was known at compilation and therefore does no type checking at run time[4]. In particular, a function with specification

```
val prtAccept: dest_port -> 'a
```

would compromise type safety; the compiler's type checker could not guarantee the correct evaluation of expressions contaminated by the function's result. In DML,

[3] An ordering preserve does not force its port groups to have a specific ordering. Orderings are assigned by port group, and ordering restrictions are enforced within preserves. For instance, if $d_1$ and $d_2$ are sent through two different totally ordered port groups that share an ordering preserve, they will arrive in the same order (as determined by ordered events and the `resolve` operator, see Section 4.6) at their intersecting destinations.

[4] Even if the data has a *dynamic* (or self-describing) type [LM91], such functions need to be aware of it so they do not confuse dynamic types with other non-dynamic SML types.

both source ports (type `'a src_port`) and destination ports (type `'a dest_port`) are parameterized by the types of their data and are created relative to `'a grp_type` values. The `'a` parameter is the type of data normally sent through the group and corresponds to the `'a` in CML's `'a chan` type. Group type values also contain the marshalling and unmarshalling routines needed to tranfer data between machines. DML provides a number of predefined group types, including `dml_string` for transmitting strings, and others can be defined by the user.

Source ports are created with the `mkSrc` function, which takes a port colony, a port context, and a group type and returns both a source port and a groupview. The returned groupview is the latest snapshot of the port group and includes the new source port. The `mkDest` function is similar to `mkSrc` but creates a destination rather than a source port. Both functions raise the `WrongType` exception if a port is being created relative to the context `VIEW gview` and the group type of `gview`'s port group does not match the one specified[5]. Sometimes a groupview will become too old and can no longer be used in port contexts. If an attempt is made to create a new port relative to such a context, `mkSrc` and `mkDest` raise the `StaleView` exception. The `rmSrc` and `rmDest` routines can be used to remove a port from port space[6].

Port identifiers—values of type `port_id`—globally distinguish ports and, unlike ports themselves, are transmissable. Port identifiers can be compared with the `sameID` comparator, and the `srcID` and `destID` functions return the identifier of a source or a destination port, respectively. Meta events of type `meta_event` describe changes to a port group's membership. For instance, the value `CREATE id` describes the creation of port $P_1$ identified by `id`. The event `REMOVE id` relates the removal of the port $P_2$ identified by `id`. If $P_3$ has port identifier `id`, the value `FAIL id` describes the failure of $P_3$'s port colony.

[5]This error is caught by assigning each group type value an index and passing the indices around in groupviews. DML's type safety relies on the uniqueness of group type indices.

[6]Applications can request that ports be garbage collected by using SML/NJ's finalization routines.

The `mkMeta` routine creates a meta port. It takes a port colony and a groupview and returns an element of type

```
(meta_event * 'la gview) Ports.dest_port
```

The destination port acts as a stream of meta event and groupview values; the meta events describe changes to the port's port group membership, which is enscapsulated by the groupviews. The meta event stream begins at the point defined by the groupview given to `mkMeta`, so applying `mkMeta` to the first groupview of a port group returns a stream of all changes to the group since it was created. To conserve memory, `mkMeta` can raise the `StaleView` exception if presented with a very old groupview.

The following expression returns a source port and destination port pair that belong the same port group and port colony:

```
let
  val clny = mkClny ();
  val (src, view) = mkSrc (clny, NEW_CONTEXT vsync, dml_string, ());
  val (dest, _) = mkDest (clny, VIEW view, dml_string, ());
in
  (src, dest)
end;
```

The code below defines a function that shuts down CML (hence DML) if any failure occurs to its group parameter:

```
fun watch (view, gtype) = let
  val meta = mkMeta (mkClny (), view, gtype);
  fun process (FAIL _) = CML.shutdown ()
    | process _ = process (portAccept meta);
in
  spawn (fn () => process (portAccept meta))
end;
```

The `portAccept` routine is like CML's `accept`: it blocks until it can return a value from its destination port argument.

## 4.3 Sending Data

After a source port has been created with the `mkSrc` command, data can be sent to its port group with `portSend`.

```
val portSend: '1a src_port * '1a -> '1a gview
```

This command takes a source port of a group $G$ and a value $v$; its return value, a `gview` $V$, is a view of $G$ that contains all destination ports that can receive $v$. In fact, each member of $V$ will either receive $v$, fail, or be removed from $G$: as we will see in Section 5.3, $V$ can be used with explicit acknowledgments to determine the exact destination set of $v$.

Although communication in DML is asynchronous, the `portSend` function can block. In fact, an invocation of `portSend` will not return until its transmitted value is *stable*, that is, guaranteed to be delivered to all extant destinations even if its sender and any subset of its recipients fail. The `portTransmit` routine is used for true asynchronous sends.

```
val portTransmit: 'a src_port * 'a -> 'a gview event
```

The `'a gview event` returned when sending a value $v$ with `portTransmit` can be used in a CML `select` or `sync` call to determine $v$'s destination set—its `event` is ready for synchronization when $v$ becomes stable. Unlike CML's `transmit`, which does nothing but set up an event value for `select`, `portTransmit` actually initiates a communication.

## 4.4 Orderings

As discussed in Sections 1.3 and 4.3, communication in DML is asynchronous, and threads do not block when they place data on source ports. Although this design decision could promote complicated programs by increasing the number of possible

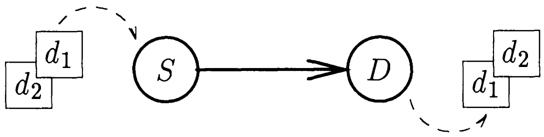



Figure 4.3: Violating fifo order

event interleavings in the system, port groups in DML are *ordered*, and some interleavings are disallowed by the system. These port group orderings also restrict the ways in which data from multiple source ports can be interleaved at common destination ports.

For instance, Figure 4.3 is a picture of communication between a source port $S$ and a destination port $D$ which share a port group $G$. Some thread placed data $d_1$ and $d_2$ on $S$ and, because the thread did not wait for $d_1$ to arrive at $D$ before sending $d_2$, $d_2$ arrived before $d_1$ at $D$. This might happen if, *e.g.*, the system used an unreliable protocol for transmitting messages between machines and $d_1$ was lost during the first attempt to send it. If $d_2$ were transmitted correctly on the first try, a later attempt to retry $d_1$ may have caused it to appear at $D$ after $d_2$. If $G$ were *fifo-ordered*, this execution history would be impossible and $d_2$ would appear after $d_1$ at $D$—DML's runtime system would hold $d_2$ until $d_1$ became available. Any two pieces of information sent from the same source port of a fifo-ordered group will be delivered in the order they were sent.

For some applications, fifo ordering is not enough to ensure correctness, and DML provides orderings with stronger constraints. Figure 4.4 shows two source ports, $S_1$ and $S_2$, and two destination ports, $D_1$ and $D_2$, that belong to the same port group $G$. The dotted line represents an instance of *thread-based ordering* in which a thread received $d_1$ from $D_1$ and then placed $d_2$ on $S_2$. Even if $G$ were fifo-ordered, the ordering of $d_1$ and $d_2$ at $D_2$ is not constrained, and, indeed, $d_2$ arrived before $d_1$

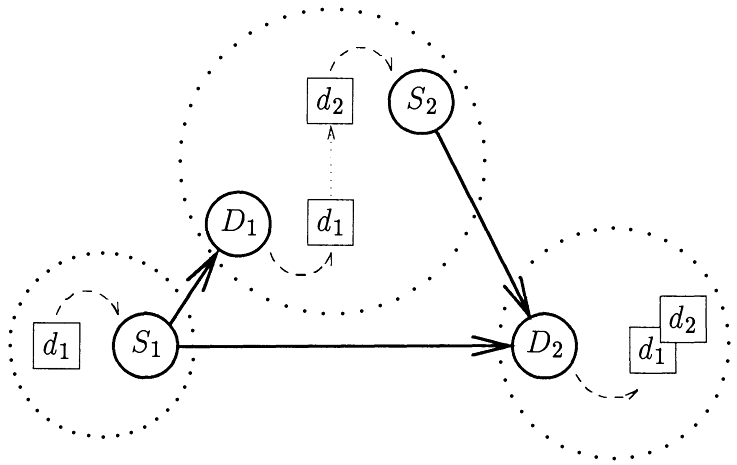



Figure 4.4: Violating causality

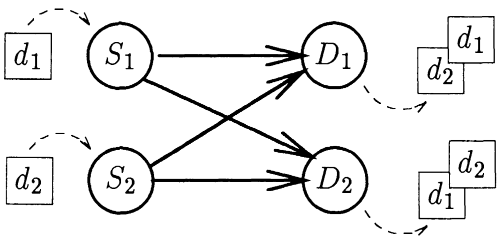



Figure 4.5: Violation of total order

at that destination port. This might have caused a problem because $d_2$ might, in some application-dependent way, have depended on $d_1$. For instance, $d_1$ may be the database operation "add ugh", and $d_2$ may have "delete ugh"; if $d_2$ is received before $d_1$ at the database, it would not behave as expected. If $G$ were *causally-ordered*, this execution history would have been impossible and $d_2$ would appear after $d_1$ at $D_2$. A group that is causally-ordered is both fifo-ordered and respects Lamport's causality relation [Lam78], where multicasts are treated as a single message transmission.

For some applications, data placed concurrently on multiple source ports must be ordered identically at each destination port. For instance, a replicated database requires that all updates be processed in the same order by all replicas. In Figure 4.5,

$d_1$ and $d_2$ were placed simultaneously on $S_1$ and $S_2$, respectively. Because their relative positions at $D_1$ and $D_2$ were not constrained (and could not be constrained with either fifo or causal ordering), $d_1$ and $d_2$ appeared in different orders at the two destination ports. This would not be acceptable if, for instance, $D_1$ and $D_2$ belonged to different copies of a replicated database and $d_1$ and $d_2$ were updates to the database, because the databases would become inconsistent with each other. If the ports had belonged to a *totally-ordered* port group, this execution history could not have taken place, and $d_1$ and $d_2$ would appear in the same order at $D_1$ and $D_2$. The message streams defined by destination ports belonging to a totally-ordered port group are identical.

## 4.5 Forms of Ordering

In Section 4.4 we tried to motivate the need for ordering constraints in a DML-like language with asynchronous and/or group-based communication primitives. However, we did not formally define what we meant by ordering; instead, we informally mentioned thread-based ordering and reasoned that other data were ordered as they appeared at destination ports. In general, there are three distinct forms of ordering in the DML system: thread-based ordering, explicit delivery ordering, and implicit delivery ordering.

### 4.5.1 Thread-based Ordering

Thread-based ordering is determined by application threads that use ports to transfer information. Two communication events (data placed on a source port or received from a destination port) are ordered in this sense if a thread executes one of them before the other. We saw in Section 4.4 that causally-ordered port groups must be aware of thread-based orderings.

Thread-based ordering can only relate communication events if their respective ports are in the same colony. That is, if a thread receives $d_1$ from $D$ and then places $d_2$ on $S$, the two events are related by a thread-based ordering only if $D$ and $S$ are in the same port colony.

It may seem strange that port colonies are used both as a unit of failure (see Section 4.1) and as a limit on thread-based orderings. However, the architecture of most distributed computations consist of independent *thread groups*: sets of threads that communicate outside of themselves only through ports (and not, for instance, through CML channels). Thread groups can be all threads that live on the same machine, within the same Unix process, or even within some finer division inside a single address space. A port colony is intended to be the set of ports that serve a single thread group. As such, a port colony is the logical unit of failure and it makes sense to ignore thread-based orderings that span colonies.

### 4.5.2 Explicit Delivery Ordering

*Explicit delivery* ordering is determined by the message stream that consitutes a destination port. While discussing the constraints on delivery ordering in the examples of Section 4.4, we were actually deliberating the ports' explicit delivery ordering.

### 4.5.3 Implicit Delivery Ordering

*Implicit delivery* ordering is ordering between multiple destination ports. Even though the examples in Section 4.4 only involved one port group, we mentioned in Section 4.1 that ordering properties are enforced across ordering preserves, and ordering preserves consist of multiple port groups. In DML, threads can *resolve* multiple destination ports and retrieve data in an order consistent with their implicit delivery ordering. In a sense, this ordering is the explicit delivery ordering of a data

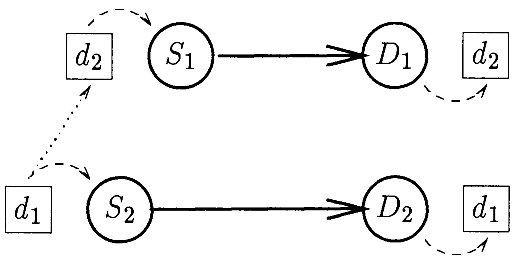



Figure 4.6: Multigroup fifo ordering

stream obtained by merging the multiple destination port data streams.

In Figure 4.6, source port $S_1$ and destination port $D_1$ are in port group $G_1$, while $S_2$ and $D_2$ are in $G_2$. As indicated by the dotted arrow, a thread placed $d_1$ on $S_1$ and then placed $d_2$ on $S_2$ (as mentioned in Section 4.5.1, $S_1$ and $S_2$ must be in the same port colony). If $G_1$ and $G_2$ are in the same ordering preserve and are both fifo-ordered, the receptions of $d_1$ at $D_2$ and $d_2$ at $D_1$ are related by an implicit delivery ordering, $d_1$ before $d_2$. It is still possible for applications to read $d_2$ first (by ignoring $D_1$), but programs that read from both $D_1$ and $D_2$ get $d_1$ first.

## 4.6 Ordered Events

As discussed in Section 3.2, the CML `receive` routine returns an `event` object that represents the potential (or ability) to accept a value from a channel. Events can describe blocking operations without committing to them, and CML uses events as input to `select`, its general selective synchronization mechanism. Presented with a list of `events`, `select` determines the first of them that is ready for synchronization and commits its operation, returning the result. If more than one of its arguments are ready simultaneously, it nondeterministically chooses between them.

Unfortunately, the nondeterminism of `select` makes it inappropriate to describe

```
signature ORD_EVENT = sig

  structure CML: CONCUR_ML

  type 'a ord_event

  val bottom: 'a -> 'a ord_event
  val top: 'a -> 'a ord_event

  val decide: 'a ord_event list -> 'a

  val resolve: 'a ord_event list -> 'a ord_event
  val wrap: 'a ord_event * ('a -> 'b) -> 'b ord_event

  val order: 'a CML.event -> 'a ord_event
  val unord: '1a ord_event -> '1a CML.event

  ...
end
```

Figure 4.7: The `ORD_EVENT` signature

a destination port receive operation with events. Suppose, for instance, that values $v_1$ and $v_2$ have reached destination ports $D_1$ and $D_2$, respectively, and that $v_1$ precedes $v_2$ in some implicit ordering as discussed in Section 4.5.3. CML's select is, of course, unaware of orderings, and, if used to accept from $D_1$ or $D_2$, `select` could choose $v_2$. DML provides an extended form of selective communication and *ordered events* which resemble CML's primitives but respect implicit orderings.

Structures that implement ordered events have the signature `ORD_EVENT` in Figure 4.7. An ordered event is, like an event, the ability to commit to some operation. However, the operations described by ordered events return elements of a global partial order $O$. For instance, the ordered events returned by `bottom` are always ready for synchronization, and the value they return is ordered no greater than other values in $O$. Similarly, an ordered event returned by `top` yields, when synchronized, an element ordered at least as great as other values in $O$.

The `portReceive` function has specification

```
val portReceive: '1a dest_port -> '1a ord_event
```

The ordered event returned by `portReceive` is the potential to receive a value from the specified destination port. This value is a member of the partial order $O$ and is related to other data values by the delivery orderings assigned by DML. The `decide` function is used for selective communication on ordered events. The expression

```
decide [portReceive dp1, portReceive dp2];
```

returns the least value in $O$ that becomes available on destination ports $dp_1$ or $dp_2$. This mechanism will *never* return a value $v_1$ from, say, $dp_1$, if eventually a value $v_2$ less than $v_1$ in $O$ could appear on $dp_2$. For instance, the expression

```
decide [top 0, portReceive dp]
```

will never return the top zero value—even though it is constantly available for synchronization—as long as it is possible to receive a value from $dp$. Any value encountered on a destination port is necessarily less than the top value in $O$. Also,

```
decide [bottom 0, portReceive dp]
```

will always return the bottom zero value because anything on a destination port is greater than bottom in $O$. The expressions

```
decide [bottom 0, bottom 1]
decide [top 0, top 1]
```

are equivalent; they do not block and can potentially return either zero or one.

The `ORD_EVENT` signature's `resolve` and `wrap` functions can be used to create ordered synchronous abstractions that have the same rights and abilities as those returned by `portReceive`. For instance, the ordered event

```
resolve [portReceive dp1, portReceive dp2]
```

behaves as a stream of $dp_1$ and $dp_2$ values merged in a way that respects the DML's implicit delivery ordering. This ordered event can be passed to another function without revealing the stream's two destination port implementation. The `order` and `unord` functions are used to convert CML events into ordered events and vice versa, respectively. A value represented by an ordered CML event is less than a `top` value and greater than a `bottom` value but is not related to other elements of DML's partial order $O$.

# Chapter 5

# Programming Paradigms and Examples

This chapter presents several examples that illustrate the DML syntax and properties introduced in Chapter 4[1]. The first example is a distributed substitution protocol that maintains the consistency of a lambda expression in a parallel environment; it shows how DML can be used to encapsulate a communication pattern—in this instance, totally ordering group communication—that is easily integrated into typical SML code. The second example is an implementation of the popular RPC Multicast paradigm, and it shows how fault-tolerant services can be automatically derived by adding a DML communication framework to preexisting SML code. The third example illustrates how first-class functions can be used to conveniently process the results of an RPC Multicast. The fourth example presents a way to automatically increase the fault tolerance of a distributed application by replacing point-to-point channels with port groups.

[1]Part of this chapter is joint work with Robert Cooper.

```
signature DIST_SUBST = sig

  structure DML: DISTRIBUTED_ML

  datatype exp = CONST of string
               | VAR of string          (* var x *)
               | APP of exp * exp       (* E F *)
               | ABS of string * exp;   (* lambda x.E *)

  (* makeExpServer takes a totally-ordered port group
     (supporting atomic multicasts but not flushed membership
      changes), an initial expression, and the number of
     required initial servers *)
  val makeExpServer:
    (string * exp) gview *
    (string * exp) grp_type * exp * int -> unit

end;
```

Figure 5.1: Distributed substitution interface

## 5.1 Distributed Substitution

Given $n$ machines and a common lambda expression [Bar84], we would like a program that allows each machine to compute and to perform simultaneous substitutions while maintaining the consistency (*i.e.*, identical copies) of the expression. If two machines compute substitutions for the same variable and try to disseminate their work simultaneously, the replicated copies could become inconsistent if the substitutions are processed in different orders at different machines. However, if machine $m_1$ initiates the substitution $[x := y]$ at the same time that $m_2$ initiates $[x := z]$ and both $m_1$ and $m_2$ use DML's totally ordered multicast to announce their substitutions, all servers will perform the substitutions in the same order, and each copy of the expression will remain identical.

The interface to a DML program that solves this particular problem is in Figure 5.1. The `DIST_SUBST` signature provides a type `exp` to build lambda expressions and a makeExpServer function to create expression server threads. Expression server threads compute substitutions and disseminate them to all other expression server threads in the system. To build such a thread, we must have a port group to send substitutions (which have type string * exp), a group type object corresponding to subsititions, an initial expression, and an integer—the number of required initial servers. For correctness, the port group should be totally ordered and must support atomic multicasts. The former requirement can be specified when port group are created; the latter is guaranteed by all DML port groups.

Note that multiple expression servers can exist in the same DML environment, on the same machine but in different environments, or on completely different machines; however, they must all be initialized with the same port group. We present a structure with signature DIST_SUBST.

```
structure DistSubst: DIST_SUBST = struct

  structure DML = DML;
  structure CML = DML.CML;
```

The first section of code in our structure defines lambda expressions and the function subst, used to perform substitutions on them. This algorithm is standard; see, *e.g.*, pp. 420–421 of [Rea89].

```
  datatype exp = CONST of string
               | VAR of string
               | APP of exp * exp
               | ABS of string * exp;

   local
     val last = ref 0;
   in
    (* a smarter implementation of newname would allow concurrent
       invocations *)
    fun newname () = (inc last; "x" ^ (makestring (! last)));
  end;

  fun freevars (CONST _) = []
    | freevars (VAR s) = [s]
```

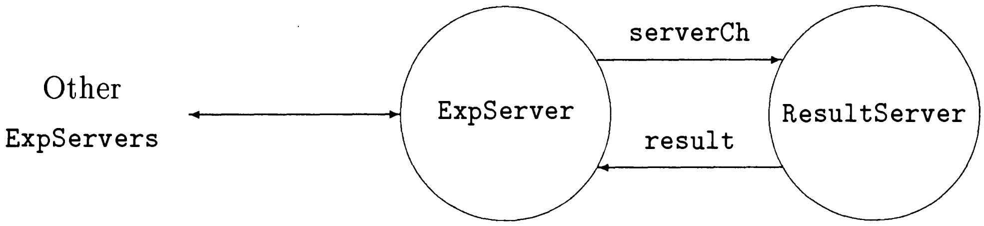



Figure 5.2: Expression server architecture

```
  | freevars (APP (x, y)) =
    ListSeq.remove_dupicates (freevars x @ (freevars y));
  | freevars (ABS (s, x)) =
    ListSeq.remove (s, freevars x, []);

(* subst (x, v, e): x [v := e], i.e., substitute e for v in x *)
fun subst (x as CONST _, _, _) = x
  | subst (VAR x, v, e) = if x = v then e else VAR x
  | subst (APP (x1, x2), v, e) =
    APP (subst (x1, v, e), subst (x2, v, e));
  | subst (x as ABS (xs, xe), v, e) =
    if xs = v then x
    else if not (ListSeq.member (xs, freevars e)) orelse
            not (ListSeq.member (v, freevars xe))
           then ABS (xs, subst (xe, v, e))
    else let
      val y = newname ();
    in
      ABS (VAR y, subst (subst (xe, VAR xs, VAR y), v, e))
    end;
```

The function makeExpServer is defined in the next section; it uses the function makeResultServer (not presented here), which creates a thread that calculates new substitutions. To communicate with this thread, makeResultServer must return a CML channel and event; the former is used to send expression-change notifications, and the latter is used to receive new substitutions calculated by the thread. The architecture is illustrated in Figure 5.2.

```
fun makeExpServer (view, gtype, M, n) = let

  (* result servers are threads that compute new substitutions.
```

```
   makeResultServer returns a channel and an event: the channel
   is used to send new expressions to the server; the
   event is used to receive new substitutions. *)
fun makeResultServer () = ...
val (serverCh, result) = makeResultServer ();
```

The function need_n is used during initialization so that expression servers do not begin communicating until all of them have been created. need_n takes a number $n$ and a DML meta destination port and groupview pair. The function returns when the group has $n$ destination ports. The DML function numDests returns the number of destination ports known to its gview argument.

```
(* need_n returns when the group has n destports *)
fun need_n (n, (mdest, view)) =
  if numDests view >= n then rmDest mdest
  else need_n (n, (mdest, #2 (portAccept mdest)));
```

The following code creates the source port, destination port and result server belonging to the new expression server.

```
val clny = mkClny ();
val (src, view) = mkSrc (clny, NEW_CONTEXT (vsync TOTAL), gtype);
val (dest, view) = mkDest (clny, VIEW view, gtype);
```

Once expression servers have initialized, they repeatedly execute the function loop. In general, an expression server waits for either an announcement from its local result server that a new substitution is ready or the reception of an incoming substitution message. In the former case, the expression server multicasts it to the other servers; in the latter case, it applies the substitution and informs its result server of the change.

```
fun loop M =
  CML.select [CML.wrap (result, fn x =>
                          (portTransmit (src, x); loop M)),
              CML.wrap (unord (portReceive dest),
                         fn (x, e) => let
                           val M = subst (M, x, e);
                         in
                           CML.send (serverCh, M);
                           loop M
                         end)];
```

The last step in creating an expression server is to spawn it into the background.

```
    in
      CML.spawn (fn () => (need_n (n, mkMeta (clny, view)); loop M))
    end;
end;
```

The code above is quite short; there are several properties that the DML system provides that allows us to write an elegant and correct solution (even in the presence of failures!) to a non-trivial distributed programming problem:

1. Because the communication group used for distributed communication is *totally ordered*, DML ensures that every expression server sees substitutions in the same order. For example, if server $s_1$ initiates the substitution $[x := y]$ at the same time that $s_2$ initiates $[x := z]$, all servers will perform the substitutions in the same order, and each copy of the expression will remain identical. In this example, naïve Remote Procedure Call implementations (even with multicast) would not be sufficient because two machines could processed the concurrent substitutions in opposite orders: if they both started with $f(x)$, one could end with $f(y)$ and the other $f(z)$.

2. Because DML provides totally-ordered *membership information* (meta data) as groups change, the expression servers can begin (virtually) simultaneously as they learn that their peers have been created.

3. All messages sent through DML port groups are *atomic*, and failures cannot cause a substitution message to be delivered to some but not all surviving expression servers.

4. Because DML provides a consistent *membership service*, no expression server will ever fail to receive a substitution message (because, *e.g.*, the sender of the message "timed out" on it) and subsequently send valid substitution messages of its own. If a server is ever detected as being faulty (or "down"), the

DML system will drop all of its future messages (unless the server is explicitly restarted).

## 5.2 RPC Multicast

The *client/server model* is a common paradigm in fault-tolerant distributed programming. In this model, a group of *server* programs control a resource (*e.g.*, a database or a speedy processor) while *client* programs communicate with the servers to access the resource. Often, the servers are replicated to provide fault tolerance. Here, communication is *two-way*; that is, clients *initiate* communications with a server and servers send *reply* messages to waiting clients. If a server group has only one member, this communication style is called RPC, or *Remote Procedure Call.* If there are several servers, the style is *RPC Multicast*; often, distributed toolkits (such as Isis [BJ87]) directly support RPC Multicast. In DML, it and other related communication mechanisms are programmable.

One possible DML implementation of RPC Multicast would require a single port group shared by both clients and servers; clients would ignore messages sent by other clients. However, this solution requires each client to have a separate destination port and would be too inefficient when clients greatly outnumber servers (which is typical). Instead, we define a type `('a, 'b) rpc_group`: clients send requests of type `'a` to servers, while servers send replies of type `'b` to clients. In our implementation, clients send a groupview with their request; this groupview is used by servers to create a source port and send replies back to the client.

```
type 'a rpc_reply = 'a * port_id
type ('a, 'b) rpc_request = 'a * 'b rpc_reply gview
type ('a, 'b) rpc_group = ('a, 'b) rpc_request gview
```

Data sent through an `rpc_group` are `rpc_requests`, pairs which contain a request and a port group for replies. Replies are pairs containing reply data and a `port_id` identifying the replying server.

The function request, used by clients to initiate RPC multicasts, has type

```
val request:
  ('a, 'b) rpc_request src_port * 'a * 'b rpc_reply grp_type ->
    'b rpc_reply dest_port * ('a, 'b) rpc_request gview event
```

This function takes a source port of an `rpc_group` $G$, a request $r$, and a group type, and transmits the $r$ to the servers that have destination ports of $G$ (the group type argument could be omitted and automatically derived from the source port, but we have not presented the appropriate syntax in this thesis). The first coordinate of the pair returned by `request` is a destination port used to queue server replies; each reply contains data of type `'b` and a `port_id` of the destination port from which the request was received. The second half of the pair is a `gview event` which, when CML's `sync` is applied to it, yields the membership of the server group when the original message was delivered. This information is enough to collect replies in a number of different ways, one of which we will examine in section 5.3.

The server uses the function `service` to reply to client queries.

```
val service:
  ('a, 'b) rpc_request dest_port * 'b grp_type * ('a -> 'b) -> unit
```

The code `service (p, t, f)` registers `f` with the destination port `p`; after `service` is evaluated, incoming requests are processed automatically; replies are calculated by feeding requests to the registered function. The group type `t` is used to convert replies to a transmissable form (as with `request`, the group type could be derived from the destination port, but this would require more syntax).

The implementations of `request` and `service` are shown in Figure 5.3. At first glance, they seem to be extremely inefficient because the number of port groups required is linear in the number of requests (assuming a bounded number of servers). However, port groups with only one destination port can be implemented with point-to-point communication and are therefore cheap.

```
fun request (sp, gtype, msg) = let
    val (replies, rview) =
      mkDest (mkClny (), NO_CONTEXT (vsync FIFO), gtype);
  in
    (replies, portTransmit (sp, (msg, rview)))
  end;

fun service (dp, gtype, fcn) =
  (wrap (unord (portReceive dp), fn (msg, rg) =>
         let val (sp, _) = mkSrc (mkClny (), VIEW rg, gtype);
         in portTransmit (sp, (fcn msg, portID dp)); ()
         end);
   service (dp, gtype, fcn));
```

Figure 5.3: Implementations of `request` and `service`

## 5.3 Processing Replies

RPC Multicast clients receive a stream of reply messages and, to avoid blocking for messages that may never arrive, they must monitor the servers for failure. However, most multicast RPC clients are probably not interested in receiving a destination port and a `('a, 'b) rpc_request gview event` from their requests, even though they could synchronize on the event, use the resulting `gview` to create a meta port, and receive a notification when one of the client's servers fail. Instead, they might prefer to receive only the first reply or a list of all replies. The `procReply` function, described below, provides this information.

```
val procReply: int * ('b rpc_reply dest_port *
                      ('a, 'b) rpc_request gview event) -> 'b list
```

The `procReply` routine takes an integer $n$ and the result of a `request` call and returns a list of reply values. This list will have length no greater than $n$, but may be less than $n$ if fewer than $n$ servers reply (because $n$ is greater than the number of active servers). Its implementation is in Figure 5.4. A thread processing replies may be in one of two states, depending on whether the `gview event` returned by `request` has become ready for synchronization (*i.e.*, whether its request is stable;

```
fun procReply (n, (dp, ge)) =
  let
    fun state1 got =
      if length got = n then []
      else sync (choose [wrap (unord (portReceive dp),
                                fn (x,id) => x :: (state1 (id :: got))),
                         wrap (ge,
                                fn gv => state2 (mkMeta (clny, gv), got,
                                                 numDests gv, 0))]);
    fun state2 (mdp, got, potential, noreply) =
      if length got = n orelse
         length got + noreply = potential then []
      else
         sync (choose [wrap (unord (portReceive dp), fn (x, id) =>
                              x :: (state2 (mdp, id::got, potential, noreply)))
                       wrap (unord (portReceive mdp),
                              fn (FAIL id, _) =>
                                state2 (mdp, got, potential,
                                        if member id got then noreply
                                        else noreply + 1)
                               | _ => state2 (mdp, got, potential, noreply))]);
   in
      state1 []
   end;
```

Figure 5.4: Implementation of procReply

see Section 4.3). In the first state, it waits for either a reply or the groupview event, and, if it receives enough replies, it completes. In the second state, it uses the view returned by `request` to determine the maximum number of expected replies and waits until it receives enough replies or a reply from all remaining servers. The DML function `numDests` returns the number of destination ports known to its `gview` argument.

Some applications may need to process replies in different ways. In DML, it's easy to write variants of `procReply` that process all replies until one is received from a specific machine, or, more generally, until $n$ are received that satisify a given predicate, *e.g.*, $n$ sensor readings that are identical.

## 5.4 Replicated Processing

Suppose we are given a distributed program that communicates using only point-to-point channels (see Figure 5.5); consider how we can make it tolerate single crash failures by replicating each node [AD76]. A sketch of a non-fault-tolerant implementation of the `POINT_TO_POINT` signature appears in Figure 5.6. In the fault tolerant version we replace each node $P$ by a pair $(P, P')$ consisting of a primary node and a backup which will take over the role of the primary should it fail. The backup maintains its internal state equivalent to the primary's, and, upon the primary's failure, produces a sequence of output messages consistent with the state of the primary at the instant it crashed. To achieve this, we arrange for the backup to observe exactly the same input messages as the primary, in the same order. Where the execution of the primary is nondeterministic (*e.g.*, because of pre-emptively scheduled threads), we must ensure the backup takes the same nondeterministic decisions. We will concentrate on the ordering and atomicity properties of internode communication, ignoring other issues including how the backup would use a meta destination

```
signature POINT_TO_POINT =
sig
  signature DML: DISTRIBUTED_ML

  type 'a remote_channel
    (* Only one dest_port is permitted per channel. *)

  val mkChannel: 'a DML.grp_type -> 'a remote_channel
  val mkSrc: 'a remote_channel -> 'a DML.src_port
  val mkDest: 'a remote_channel -> 'a DML.dest_port
  val portSend: 'a DML.src_port * 'a -> unit
  val portAccept: 'a DML.dest_port -> 'a
end
```

Figure 5.5: Signature for simple point-to-point remote communication

```
functor SimpleP2P (structure DML: DISTRIBUTED_ML): POINT_TO_POINT =
  struct
    structure DML = DML;

    local open DML
    in
      type 'a remote_channel = 'a grp_type * 'a gview;

      fun mkChannel gtype =
        #2 (mkSrc (mkClny (), NEW_CONTEXT vsync, gtype));
      fun mkSrc (gtype, view) =
        #1 (mkSrc (mkClny (), VIEW view, gtype));
      fun mkDest (gtype, view) =
        #1 (mkDest (mkClny (), VIEW view, gtype));
      (* In practice there would be code to ensure only one
         dest_port per channel. *)

      fun portSend (src, data) = CML.sync (portTransmit (src, data));
      fun portAccept dest = DML.portAccept dest;
    end;
  end
```

Figure 5.6: Non-fault-tolerant implementation of point-to-point channels

port to notice the failure of the primary and how to handle nondeterminism due to thread scheduling and external I/O [BBG+89,BCG91].

We represent each point-to-point communication channel in the original program by a port group containing two destination ports, one owned by each of the primary and backup nodes (see Figure 5.7). The backup reads these messages and performs the same actions as the primary, except that output messages from the backup are suppressed. When the primary fails, the backup assumes the primary's role and sets the variable `primary` to `true`. In addition there is a point-to-point port group (`backup_chan`) from the primary to the backup. The primary uses this connection to send messages to the backup that resolve any nondeterminism caused by internal scheduling decisions.

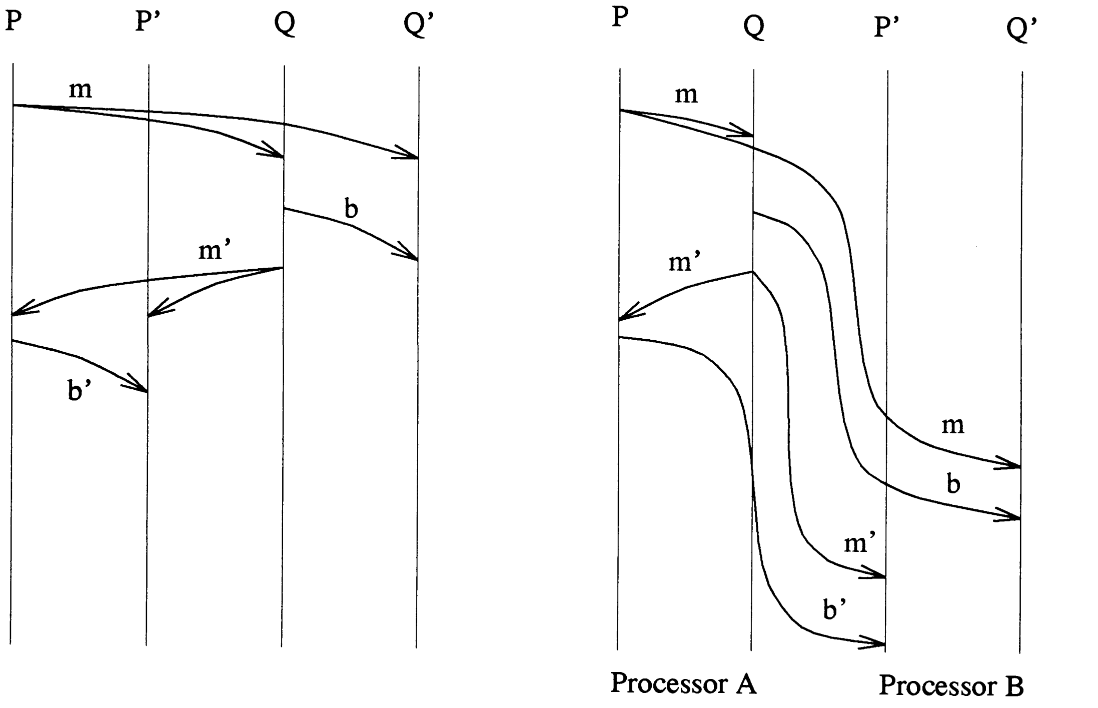
A point-to-point message, $m$, from node $P$ to $Q$ in the original program is transformed into a multicast from $P$ to $Q$ and its backup $Q'$ (see Figure 5.8). Using a totally ordered multicast, $Q$ and $Q'$ will receive this message in the same order relative to other multicasts.

We must also ensure that the scheduling message, $b$, sent by $Q$ over the backup channel is delivered at $Q'$ after the causally preceding multicast, $m$. If synchronous, non-causal communication is used, $P$ must wait when it transmits $m$ until both $Q$ and $Q'$ have received and acknowledged the message. If causally ordered communication is used throughout, $P$ can initiate the multicast and immediately continue processing while the message is still being delivered. The causal ordering protocol includes information in message $b$ identifying $m$ as a causally preceeding message that must be delivered first. With synchronous, non-causal communication, an extra message delay is inserted into the critical path of the program.

This performance penalty is more severe than it at first appears. To increase the independence of failures we might locate the primary and backup of a pair in physically separate locations. To improve performance in the normal case (no failures)

```
signature PROCESS_PAIR =
  sig
    val primary: bool ref
    type schedule (* Information about nondeterministic decisions made
                     by primary *)
    val backup_chan: schedule gview
  end

functor ResilientP2P(structure DML: DISTRIBUTED_ML
                     and ProcessPair: PROCESS_PAIR): POINT_TO_POINT =
  struct
    structure DML = DML;

    local open DML
    in
      type 'a remote_channel = 'a grp_type * 'a gview;

      fun mkChannel gtype =
        #2 (mkSrc (mkClny (), NEW_CONTEXT vsync, gtype));
      fun mkSrc (gtype, view) =
        #1 (mkSrc (mkClny (), VIEW view, gtype));
      fun mkDest (gtype, view) =
        #1 (mkDest (mkClny (), VIEW view, gtype));
      (* In practice there would be code to ensure only one dest_port
         at each of the primary and backup. *)

      fun portSend (src, data) =
        if ProcessPair.primary
          then CML.sync (portTransmit (src, data))
        else ();

      fun portAccept dest = DML.portAccept dest;
    end;
  end
```

Figure 5.7: Fault tolerant implementation of point-to-point channels




Logical message pattern

Primaries and backups on different processors

Figure 5.8: Message patterns in fault-tolerant process pairs.

we might locate all the primaries near each other (perhaps on the same computer). Now the synchronous approach performs much more slowly than the asynchronous causal approach. In the synchronous case, all communication will occur at the speed dictated by the primary-backup connections. In the asynchronous approach, communication among the primaries can proceed at close to the speed attainable in the original non-fault-tolerant program. Communication between primaries and backups can proceed slower, in the background. Because messages are asynchronous, multiple primary-backup messages can be combined automatically into a smaller number of large network packets which will make much more efficient use of the network. We are not limited by network latency (round trip packet times), but bandwidth. Bandwidth scales much better than latency on almost all network technologies.

But what happens if a primary fails before its backup node has received all of the messages destined for it? The DML runtime system ensures that for any message received by another node, the causally preceding messages destined for the backup will be delivered to it (this property is called *causal completeness*). Thus, if any "evidence" of an action taken by the primary has been observed by another node in the program, the backup will receive any prior messages from the primary. Conversely, if no evidence of the final few actions of the failed primary is visible, we can present the illusion that those actions never took place. There remains the possibility that the backup will send out some messages that duplicate the last few messages sent by the primary. These duplicates can be detected easily at the destinations using sequence numbers.

We see that asynchronous communication would not be feasable in this example without the causal ordering and completeness properties, which ensure that asynchronous messages are delivered in the correct order even when they are delayed and nodes crash, and the multicast atomicity property, which ensures that either all destinations receive the multicast, or none of them do. Asynchronous communication, coupled with appropriate ordering and failure properties, permits higher network performance than synchronous communication, and, as networks become faster relative to processors, asynchronous communication becomes essential. Additionally, DML's ordering properties, such as total and causal orderings, are vital to both asynchronous and group-based communication; they reduce the number of possible event histories—and therefore simplify the programmer's task by eliminating weird system behaviours—without compromising performance.

# Chapter 6

# A Theory of Group-Based Communication

In this chapter we define a calculus and a set of rules for reasoning about group-based asynchronous communication. We introduce a formal notation, called a *communication collection*, to describe asynchronous message conversations, and introduce several properties of collections—implementability, the preservation of order, and delivery atomicity—that are useful. We formally define *virtual synchrony* in terms of these properties and show that our definition implies definitions of virtual synchrony given elsewhere.

Although the formalisms in this chapter were developed to reason about DML programs, they can be used to describe the executions of any group-based programming system that supports atomic message delivery, including Isis [BCG91], Transis [ADKM92], and Horus [vRBGR93].

## 6.1 Communication Collections

Infomally, an execution sequence (or history) of a distributed system consists of a finite number of communication partners and a log of the messages sent between them. In this section we formalize this notion; we call an execution sequence a *communication collection* and identify each communication partner with *communication sequence identifiers.*

Formally, a communication collection $\mathcal{C}$ is a four-tuple of the form $(P, M, \mathsf{seq}, \mathsf{Init})$, where:

- $P$ is a finite set of identifiers, called communication sequence identifiers,
- $M$ is a set of *messages*—elements of the form $msg_n$, $join_p$, or $leave_p$ for $n$ an integer and $p \in P$,
- seq is a function that maps the communication sequence identifier $p \in P$ to a sequence of $p$-events, each of which have the form $\textcircled{p} \rhd m$ ($p$ sends $m$) or $\textcircled{p} \lhd m$ ($p$ delivers $m$) for $m \in M$, and
- Init is a function from $P \mapsto 2^P$.

Each communication event sequence represents the message log of a single communication partner (identified by its sequence identifier). For any given identifier $p$, $\mathsf{Init}(p)$ will be interpreted (after we introduce suitable axioms in Section 6.3) as the initial "group membership" observed by $p$.

We write $e_1 \rightarrow_C e_2$ to denote that the event $e_1$ precedes $e_2$ in some sequence of the communication collection $C$. The predicate $Ex_C(e)$ holds if $e$ belongs to one of $\mathcal{C}$'s event sequences. The event $pred_{\mathcal{C}}(e)$ is the immediate predecessor of $e$ in $\mathcal{C}$, and is undefined if $Ex_C(e)$ is false or $e$ is the first event in its sequence.

**Example 6.1.** Consider the following visualization of a communication collection

$\mathcal{C}$:

$$\mathcal{C} : \begin{cases} p: & \{\} \quad ⓟ \triangleright join_p,\ ⓟ \triangleleft join_p,\ ⓟ \triangleleft join_q,\ ⓟ \triangleright msg_0,\ ⓟ \triangleleft msg_0 \\ q: & \{p\} \quad ⓠ \triangleright join_q,\ ⓠ \triangleleft join_q,\ ⓠ \triangleleft msg_0 \end{cases}$$

We interpret this pictorial presentation as a history of two communication endpoints: $p$, and $q$. We see that $\mathsf{Init}(p) = \{\}$ and $\mathsf{Init}(q) = \{p\}$. Because $p$ received a join notice from $q$ (that is, the event $ⓟ \triangleleft join_q$ exists) and $p$ was in $q$'s inital group membership, we might say that $p$ was a member of the group before $q$.

More precisely, $p$ and $q$ could be active DML objects that use a port group $G$ to share information. Under this interpretation, the message $join_p$ was sent when $p$ created its first port, and $msg_0$ is data sent by $p$ through $G$. The object $q$ created its first port after $p$, and $p$ learned of $G$'s by receiving $join_q$ on a meta port.

Later in this chapter we will formalize the notion of *group membership* and identify some restrictions on communication collections that will let us interpret them as we have informally done above.

## 6.2 Implementability

In general, we are only interested in communication collections that can be modeled by executions of a real distributed system. However, the definition in Section 6.1 does not preclude collections that lack intuitive interpretations.

**Example 6.2**. Consider the following communication collection $\mathcal{C}$:

$$\mathcal{C} : \begin{cases} r: & \{\} \quad ⓡ \triangleleft m_2,\ ⓡ \triangleright m_1,\ ⓡ \triangleleft m_3 \\ s: & \{\} \quad ⓢ \triangleleft m_1,\ ⓢ \triangleright m_2 \end{cases}$$

In this example, $r$ delivers message $m_2$ before sending $m_1$; in any natural interpretation, this would imply that $m_1$ came into existence before $m_2$ did. But this cannot be true, because $s$ delivers $m_1$ before sending $m_2$! Also, $r$ delivers $m_3$, which was never sent: the syntax of communication collections lets us write things that do not make sense and could never be realized.

We therefore define a property so that all collections satisfying it do not exhibit the weird patterns we saw in the example above. Define the relation $\prec_{\mathcal{C}}$ by $a \prec_{\mathcal{C}} b$ ("$a$ immediately causes $b$") if and only if

1. $a \rightarrow_{\mathcal{C}} b$, or

2. There exists $m$, $p$, and $q$ in $\mathcal{C}$ for which $a = ⓟ \triangleright m$ and $b = ⓠ \triangleleft m$.

Define $\prec^*_{\mathcal{C}}$ to be the transitive closure of $\prec_{\mathcal{C}}$; $\prec^*_{\mathcal{C}}$ is Lamport's causality relation for unicasts[1] [Lam78].

Note that $a \prec^*_{\mathcal{C}} b$ if and only if there is a chain of events $a = e_0, \ldots, e_n = b$ in $\mathcal{C}$ such that, for all $i$, either $e_i$ precedes $e_{i+1}$ in some event sequence, or $e_i$ is a send event and $e_{i+1}$ is a deliver event of the same message. The relation $\prec^*_{\mathcal{C}}$ is sometimes called the "happens before" relation because $a \prec^*_{\mathcal{C}} b$ if and only if $\mathcal{I}(a)$ must exist before $\mathcal{I}(b)$ occurs in any natural interpretation $\mathcal{I}$ of $a$ and $b$ as communication events. Because we want communication collections to describe real systems, require the "happens before" relation to be a partial order.

We call a communication collection $\mathcal{C}$ *implementable* if:

- If $a \prec^*_{\mathcal{C}} b$ then $b \not\prec^*_{\mathcal{C}} a$ (implying that $\prec^*_{\mathcal{C}}$ is a non-reflexive partial order);

- If the event $ⓟ \triangleleft m$ exists for some $m$ and $p$ in $\mathcal{C}$, then there is a $q$ such that $ⓠ \triangleright m$ exists in $\mathcal{C}$.

In example 2, we have both

$$ⓡ \triangleleft m_2 \prec^*_{\mathcal{C}} ⓡ \triangleright m_1$$

and

$$ⓡ \triangleright m_1 \prec^*_{\mathcal{C}} ⓡ \triangleleft m_2,$$

[1]The $\prec^*_{\mathcal{C}}$ relation is not multicast causality: $ⓟ \triangleleft m$ and $ⓠ \triangleleft m$ for $p \neq q$ are not equated.

so $\prec_C^*$ is not a partial order. Also, the send event of $ⓡ \triangleleft m_3$ does not exist, so $\mathcal{C}$ violates both conditions for implementablity.

## 6.3 Views

We now associate a set of endpoints—called a *view*—to every event in a communication collection. This set will be interpreted as the current groupview (local to the event sequence) at that particular event.

**Example 6.3**. Consider the implementable communication collection from example 1:

$$\mathcal{C} : \begin{cases} p: & \{\} & ⓟ \triangleright join_p,\ ⓟ \triangleleft join_p,\ ⓟ \triangleleft join_q,\ ⓟ \triangleright msg_0,\ ⓟ \triangleleft msg_0 \\ q: & \{p\} & ⓠ \triangleright join_q,\ ⓠ \triangleleft join_q,\ ⓠ \triangleleft msg_0 \end{cases}$$

If $p$ and $q$ are interpreted as communication partners, each has a natural concept of the conversation (or group) participants at any point in their event streams; this set of participants we call a *view*. Because $p$'s $\mathsf{Init}$ set is empty, its view of the group at each event preceding $ⓟ \triangleleft join_p$ is empty. At $ⓟ \triangleleft join_p$ its view is $\{p\}$, and, because $ⓟ \triangleleft join_q$ immediately follows $ⓟ \triangleleft join_p$, its view at all other events is $\{p, q\}$. As for $q$, its initial view is $\{p\}$ and changes to $\{p, q\}$ when it delivers its own join message.

Formally, we define how a set of endpoints (which we intend to interpret as a view) changes relative to an event $e$:

$$Change(V, e) = \begin{cases} V \cup \{q\} & (\exists p)(e = ⓟ \triangleleft join_q) \\ V \setminus \{q\} & (\exists p)(e = ⓟ \triangleleft leave_q) \\ V & \text{otherwise} \end{cases}$$

By this definition, a set $V$ is changed by the delivery of join and leave events but is not affected by any other kind of event. The operator $\lessdot_{\mathcal{C}}$ (pronounced "view"), which maps events of the communication collection $\mathcal{C} = (P, M, \mathsf{seq}, \mathsf{Init})$ to subsets of $P$ (*i.e.*, communication endpoints), by:

$$\lessdot(e) = \begin{cases} \{\} & \neg Ex(e) \\ \mathit{Change}(\mathsf{Init}(p), e) & e = \mathit{first}(p) \\ \mathit{Change}(\lessdot(\mathit{pred}(e)), e) & \text{otherwise} \end{cases}$$

The view of an event $e$ that belongs to $p$'s event sequence is determined by $\mathsf{Init}(p)$, the structure of $e$, and the structure of $e$'s predecessors. Views as defined here are local to event sequences; we have not specified any view consistency properties across communication collections[2]. We refer to messages of the form $join_p$ and $leave_p$ as *view change* messages. As mentioned in Section 6.1, we interpret $\mathsf{Init}(p)$ as the initial group membership observed by $p$.

## 6.4 Delivery Atomicity

In most group-based communication systems [BJ87,PBS89,LLS90,vRBGR93], messages destined for a group are delivered at each of the group's members. In this section, we formulate a property of communication collections that can be used to describe the event sequences produced by these systems.

In Section 6.3, we defined the operator $\lessdot$ and interpreted views as the set of endpoints thought to be in the group at each event in a communication collection. To formalize a property like "each member of the group eventually delivers message $m$," we need a collection-wide definition of membership defined on all delivery events. There are at least two reasonable ways to do this:

1. We could define an endpoint $p$ to be a member of $p$'s collection at ⓠ $\triangleleft\, m$ if and only if ⓟ $\triangleleft\, m$ exists and lies between ⓟ $\triangleleft\, join_p$ and ⓟ $\triangleleft\, leave_p$, *i.e.*, in the interval between $p$'s local deliveries of its own join and leave. This definition

[2] In particular, views are merely sets and are not *sequenced* as in [Ste91] and [Ric92]. View equality is therefore strictly weaker here than in those papers, and consistency property definitions must be modified accordingly.

makes membership a local property because $p$'s membership is determined by $p$'s own events.

2. We could define $p$ to be a member of its collection at $\textcircled{q} \triangleleft m$ if and only if $p \in \lessdot(\textcircled{q} \triangleleft m)$. This definition makes $p$'s membership dependent on the sequencing of events elsewhere.

We avoid deciding between these definitions (and others) by removing membership from our definition entirely. Instead, we make the strange requirement that every endpoint to deliver all messages—if every endpoint delivers everything, then, for all messages $m$ and any definition of "member," we know that "each member of the group eventually delivers $m$."

A communication collection $\mathcal{C} = (P, M, \mathsf{seq}, \mathsf{Init})$ is *delivery atomic* if, for all $p \in P$ and $m \in M$,

$$Ex_{\mathcal{C}}(\textcircled{p} \triangleright n) \Rightarrow (\forall q)(\textcircled{q} \triangleleft m),$$

*i.e.*, every message that is sent is delivered everywhere. At first, this seems to be a silly definition; it appears that delivery atomic collections could not describe efficient implementations that support dynamic group memberships. Such systems would be forced to deliver all messages to potential members of every group—a task which may not be possible if group members can be reconfigured dynamically.

In general, we would only expect an implementation to provide a subset of the events that form a delivery atomic communication collection. However, we want to constrain implementations so that events generated by them (interpreted as communication collections) have a natural *view preserving* supercollection that is delivery atomic. Such extendible collections are called *subcollections.*

## 6.5 Subcollections

Let $\mathcal{C} = (P, M, \mathsf{seq}, \mathsf{Init})$ and $\mathcal{C}' = (P', M', \mathsf{seq}', \mathsf{Init}')$ be two communication collections. $\mathcal{C}$ is a *subcollection* of $\mathcal{C}'$ if

1. $P \subseteq P'$ and $M \subseteq M'$;
2. For each $p \in P$, there are event sequences *pref* and *suff* such that the concatenation of *pref*, $\mathsf{seq}(p)$, and *suff* is the same as $\mathsf{seq}'(p)$;
3. For each event $e$ in $\mathcal{C}$, $\lessdot_{\mathcal{C}}(e) = \lessdot_{\mathcal{C}'}(e)$.

This last condition constrains $\mathcal{C}$'s Init function; it is equivalent to saying that, for each $p \in P$, either

$$\mathit{first}_{\mathcal{C}}(p) = \mathit{first}_{\mathcal{C}'}(p) \text{ and } \mathsf{Init}(p) = \mathsf{Init}'(p)$$

or

$$\mathsf{Init}(p) = \lessdot_{\mathcal{C}'}(\mathit{pred}_{\mathcal{C}'}(\mathit{first}(p))).$$

We sometimes call $C'$ a *supercollection* of $\mathcal{C}$ if $\mathcal{C}$ is a subcollection of $\mathcal{C}'$.

A supercollection $\mathcal{C}'$ of a communication collection $\mathcal{C}(P, M, \mathsf{seq}, \mathsf{Init})$ is called *non-presumptuous* if, for each $m \in M$,

$$p \in \lessdot_{\mathcal{C}'}(ⓟ \triangleleft m) \Rightarrow \mathit{Ex}_{\mathcal{C}}(ⓟ \triangleleft m).$$

That is, a delivery event ⓟ$\triangleleft m$ added by a non-presumptuous supercollection can not be interpreted as having occurred while $p$ was participating in the group conversation.

As we will see in the following sections, it is often easier to state and reason about properties of delivery atomic collections than their non-delivery atomic subcollections. However, this fact would be useless if observable subcollections cannot be extended: we will show in Chapter 7 that the event sequences generated by a DML program can always be interpreted as a subcollection of a delivery atomic and non-presumptous communication collection.

## 6.6 Orderings

Many distributed protocols enforce *message orderings* that relate the juxtaposition of send and deliver events in distributed systems. For instance, TCP [Bla92], the virtual circuit protocol on the Internet, guarantees "first in, first out" (or *fifo*) delivery between pairs of sockets. Communication in programs supported by the Isis Distributed Toolkit usually respects a form of *causal* ordering [BCG91], which is a generalization of fifo delivery to multiple communication partners.

Consider a communication collection $\mathcal{C} = (P, M, \mathsf{seq}, \mathsf{Init})$. Given $M' \subseteq M$, we define four subsets of $M$ called *ordering classes*: $\mathit{fifo}_{\mathcal{C}}(M')$, $\mathit{causal}_{\mathcal{C}}(M')$, $\mathit{total}_{\mathcal{C}}(M')$, and $\mathit{flushed}_{\mathcal{C}}(M')$. The classes are not in general hierarchical, *e.g.*, an element of $\mathit{causal}_{\mathcal{C}}(M')$ need not be in $\mathit{fifo}_{\mathcal{C}}(M')$ or vice versa.

A message $n \in M$ is in the set $\mathit{fifo}_{\mathcal{C}}(M')$ if and only if, for all $m \in M' \subseteq M$,

$$(\exists p)(ⓟ \triangleright m \rightarrow_{\mathcal{C}} ⓟ \triangleright n) \Rightarrow (\forall q)(ⓠ \triangleleft m \rightarrow_{\mathcal{C}} ⓠ \triangleleft n)$$

Although this definition is well-defined for general $\mathcal{C}$, it has its intended meaning only if $\mathcal{C}$ is delivery atomic. We will discuss this point again later in this section.

A message $n \in M$ is in $\mathit{causal}_{\mathcal{C}}(M')$ if and only if, for all $m \in M' \subseteq M$,

$$(\exists p)(ⓟ \triangleleft m \rightarrow_{\mathcal{C}} ⓟ \triangleright n) \Rightarrow (\forall q)(ⓠ \triangleleft m \rightarrow_{\mathcal{C}} ⓠ \triangleleft n).$$

A message $n \in M$ is in $\mathit{total}_{\mathcal{C}}(M')$ if and only if, for all $m \in M' \subseteq M$,

$$(\exists p)(ⓟ \triangleleft m \rightarrow_{\mathcal{C}} ⓟ \triangleleft n) \Rightarrow (\forall q)(ⓠ \triangleleft m \rightarrow_{\mathcal{C}} ⓠ \triangleleft n).$$

Although a message in $\mathit{causal}_{\mathcal{C}}(M')$ need not be in $\mathit{fifo}_{\mathcal{C}}(M')$, we show below that $m \in \mathit{total}_{\mathcal{C}}(M') \Rightarrow m \in \mathit{causal}_{\mathcal{C}}(M')$ if $\mathcal{C}$ is implementable and delivery atomic.

A message $n \in M$ is in $\mathit{flushed}_{\mathcal{C}}(M')$ if and only if, for all $m \in M' \subseteq M$,

$$(\exists p)(ⓟ \triangleright m \rightarrow_{\mathcal{C}} ⓟ \triangleleft n) \Rightarrow (\forall q)(ⓠ \triangleleft m \rightarrow_{\mathcal{C}} ⓠ \triangleleft n).$$

**Theorem 1.** If $\mathcal{C} = (P, M, \mathsf{seq}, \mathsf{Init})$ is implementable and delivery atomic, and if $M' \subseteq M$, then $total_{\mathcal{C}}(M') \subseteq causal_{\mathcal{C}}(M')$.

**Proof.** If $total_{\mathcal{C}}(M') = \{\}$, the theorem is vacuously true. Otherwise, let $n \in total_{\mathcal{C}}(M')$ and suppose, for some $m \in M'$,

$$(\exists p)(\textcircled{p} \triangleleft m \rightarrow_{\mathcal{C}} \textcircled{p} \triangleright n).$$

Because $\mathcal{C}$ is delivery atomic, $Ex(\textcircled{p} \triangleleft n)$; because $\mathcal{C}$ is implementable (and hence $\prec^*_{\mathcal{C}}$ is a partial order), $\textcircled{p} \triangleright n \rightarrow_{\mathcal{C}} \textcircled{p} \triangleleft n$. Thus

$$(\exists p)(\textcircled{p} \triangleleft m \rightarrow_{\mathcal{C}} \textcircled{p} \triangleleft n),$$

and, because $n \in total_{\mathcal{C}}(M')$,

$$(\forall q)(\textcircled{q} \triangleleft m \rightarrow_{\mathcal{C}} \textcircled{q} \triangleleft n).$$

Therefore $n \in causal_{\mathcal{C}}(M')$, and $total_{\mathcal{C}}(M') \subseteq causal_{\mathcal{C}}(M')$. $\square$

**Theorem 2.** If $\mathcal{C} = (P, M, \mathsf{seq}, \mathsf{Init})$ is implementable and delivery atomic, and if $M' \subseteq M$, then $flushed_{\mathcal{C}}(M') \subseteq fifo_{\mathcal{C}}(M')$.

**Proof.** If $flushed(M') = \{\}$, the theorem is vacuously true. Otherwise, let $n \in flushed_{\mathcal{C}}(M')$ and suppose, for some $m \in M'$,

$$(\exists p)(\textcircled{p} \triangleright m \rightarrow_{\mathcal{C}} \textcircled{p} \triangleright n).$$

Because $\mathcal{C}$ is delivery atomic, $Ex(\textcircled{p} \triangleleft n)$; because $\mathcal{C}$ is implementable (and hence $\prec^*_{\mathcal{C}}$ is a partial order), $\textcircled{p} \triangleright n \rightarrow_{\mathcal{C}} \textcircled{p} \triangleleft n$. Thus

$$(\exists p)(\textcircled{p} \triangleright m \rightarrow_{\mathcal{C}} \textcircled{p} \triangleleft n),$$

and, because $n \in flushed_{\mathcal{C}}(M')$,

$$(\forall q)(\textcircled{q} \triangleleft m \rightarrow_{\mathcal{C}} \textcircled{q} \triangleleft n).$$

Therefore $n \in fifo_{\mathcal{C}}(M')$, and $fifo_{\mathcal{C}}(M') \subseteq flushed_{\mathcal{C}}(M')$. $\square$

It is fairly easy to provide examples of, *e.g.*, messages in $causal_{\mathcal{C}}(M')$ but not in $fifo_{\mathcal{C}}(M')$ and messages in $flushed_{\mathcal{C}}(M')$ but not in $total_{\mathcal{C}}(M')$. However, many systems that provide ordering primitives enforce other relationships anyway. In Isis, for instance, $causal_{\mathcal{C}}(M') \subseteq fifo_{\mathcal{C}}(M')$, and in DML, $flushed_{\mathcal{C}}(M') \subseteq total_{\mathcal{C}}(M')$.

Our delivery atomicity assumption has made it easier to define ordering in communication collections, and we will not define ordering for non-delivery atomic collections in this thesis. Indeed, finding a definition that matches our intutive notion of ordering in non-delivery atomic collections is difficult. For instance, suppose we have a system that always *attempts delivery*—*i.e.*, messages sent to a group will be delivered to each member which stays in the group for a sufficient amount of time. Consider the following collection:

$$\mathcal{C} \approx \begin{cases} p: & \{p,q\} \quad ⓟ \triangleright m,\ ⓟ \triangleright n,\ ⓟ \triangleleft m,\ ⓟ \triangleleft n,\ ⓟ \triangleleft leave_q, \\ q: & \{p,q\} \quad ⓠ \triangleleft n,\ ⓠ \triangleright leave_q,\ ⓠ \triangleleft leave_q \end{cases}$$

Using the definition above, $n \in fifo_{\mathcal{C}}(\{m\})$; however, because our system attempts delivery, $m$ would have been delivered to $q$ if $q$ had not sent $leave_q$. Had this occurred, we would have had $ⓠ \triangleleft n \rightarrow_{\mathcal{C}} ⓠ \triangleleft m$, and then $n \notin fifo_{\mathcal{C}}(\{m\})$. In such a system, then, we need a different (and probably more complex) definition to capture our intuitive notion of order.

Another (more restrictive) way to define order would be to retain the delivery atomic definitions but require all messages delivered before a *join* or *leave* message $m$ to be delivered everywhere before $m$. Such "view change" messages would then form well-defined cuts in each event sequence, and strange interactions between membership changes and ordinary messages, as illustrated above, would not occur. This approach, called *virtual synchrony*, is that taken by Isis [BJ87,BCG91]; it is equivalent to requiring *join* and *leave* messages to be in $total_{\mathcal{C}}(M)$[3].

[3] It may be necessary for messages sent by leaving (or failing) entities to be delivered everywhere before the appropriate leave notice. In that case, the latter must be in $fifo_{\mathcal{C}}(M)$ as well as in $total_{\mathcal{C}}(M)$.

## 6.7 Virtual Synchrony

Although virtual synchrony has been shown to be an excellent model for building replicated distributed services [BC90], it has not been easy to define. Birman, Schiper, and Stephenson [BSS91] do not give a formal definition of virtual synchrony at all, and, as we will show in Section 6.7.3, the one in Stephenson's doctoral thesis [Ste91] is too weak: it labels as virtually synchronous certain communication patterns which we understand to be inherently non-virtually synchronous. We show in Section 6.7.3 that our definition implies Stephenson's[4].

Formally, a communication collection $\mathcal{C}$ is *virtually synchronous* if it is implementable and has a non-presumptous supercollection $\mathcal{C}' = (P', M', \mathsf{seq}', \mathsf{Init}')$ such that:

1. for all messages $m = join_p \in M'$, $m \in total_{\mathcal{C}}(M)$.
2. for all messages $n = leave_p \in M'$, $n \in fifo_{\mathcal{C}}(M) \cap total_{\mathcal{C}}(M)$;
3. for all $p \in P'$, $\mathsf{Init}'(p) = \{\}$.

Virtual synchrony is primarily a constraint on the order in which join and leave message delivery events appear in their event sequences. Such collections have several nice properties, three of which we present in the theorems below.

### 6.7.1 Indexable Views

The views of a communication collection $\mathcal{C} = (P, M, \mathsf{seq}, \mathsf{Init})$ are *indexable* if:

1. there is an injective mapping index from $\mathcal{C}$'s view change messages to the integers such that, for each $p$ in $P$, the application of index to the sequence of view change messages delivered by $p$ is a series of consecutive, increasing integers;

[4]All known protocols that claim to be virtually synchronous, including Stephenson's and the original Isis protocol [BJ87], satisfy our definition.

2. for each view change message $m$ and $p, q \in P$,

$$\lessdot(\textcircled{p} \lhd m) = \lessdot(\textcircled{q} \lhd m).$$

We say that the function $\mathsf{index}$ *indexes the views of* $\mathcal{C}$.

**Example 6.4.** Consider the communication collection

$$\mathcal{C} : \begin{cases} p: & \{\} \quad \textcircled{p} \rhd join_p,\ \textcircled{p} \lhd join_p,\ \textcircled{q} \lhd join_q \\ q: & \{\} \quad \textcircled{q} \lhd join_p,\ \textcircled{q} \rhd join_q,\ \textcircled{q} \lhd join_q \end{cases}$$

In this example, the view changes messages are $join_p$ and $join_q$. Define the function $\mathsf{index}$ by

$$\mathsf{index}(join_p) = 1 \text{ and } \mathsf{index}(join_q) = 2.$$

Certainly $\mathsf{index}$ is injective. Moreover, the sequence of view changes messages delivered by both $p$ and $q$ is $[join_p, join_q]$, and the application of $\mathsf{index}$ to this sequence yields $[1, 2]$, a series of consecutive, increasing integers. Also,

$$\lessdot(\textcircled{p} \lhd join_p) = \lessdot(\textcircled{q} \lhd join_p) = \{p\},$$

and

$$\lessdot(\textcircled{p} \lhd join_q) = \lessdot(\textcircled{q} \lhd join_q) = \{p, q\},$$

so $\mathsf{index}$ indexes the views of $\mathcal{C}$.

A collection with indexable views has a well-defined notion of group-wide membership. To see this, let $\mathcal{C} = (P, M, \mathsf{seq}, \mathsf{Init})$ have indexable views, and construct the message sequence $S$ by selecting all view change messages in $\mathcal{C}$ and sorting them relative to $\mathsf{index}$; because $\mathsf{index}$ is injective, there is only one way to construct $S$. By property two, $S$ defines a sequence $S'$ of views. Moreover, by property one, the sequence of views traced by the elements of $\mathsf{seq}(p)$ for all $p \in P$ must be extendible by a prefix and suffix to $S'$; in this sense, $S'$ is a global view change sequence and can be used to define a unique group-wide membership history for $\mathcal{C}$.

**Theorem 1.** If $\mathcal{C}$ is virtually synchronous, then $\mathcal{C}$'s views are indexable.

**Proof.** Let $\mathcal{C}' = (P', M', \mathsf{seq}', \mathsf{Init}')$ be a virtually synchronous and delivery atomic supercollection of $\mathcal{C}$ (perhaps $\mathcal{C}' = \mathcal{C}$). For any view change delivery event $e$ in an event sequence of $C'$, define:

$$\mathsf{chnum}(e) = |\{e' \to e \mid e' \text{ is a view change delivery event}\}|.$$

Suppose that there exists a view change message $m \in M'$ and identifiers $p, q \in P'$ such that

$$\mathsf{chnum}(\textcircled{p} \lhd m) \neq \mathsf{chnum}(\textcircled{q} \lhd m).$$

Let $S$ and $T$ be the set of all view change messages delivered before $m$ in $p$ and $q$'s event sequences, respectively. By the definition of $\mathsf{chnum}$, $|S| \neq |T|$; without loss of generality, assume $S \not\subseteq T$ and choose $s \in (S \setminus T)$. Because $\mathcal{C}'$ is delivery atomic,

$$\textcircled{p} \lhd s \to_{\mathcal{C}'} \textcircled{p} \lhd m \quad \text{and} \quad \textcircled{q} \lhd m \to_{\mathcal{C}'} \textcircled{q} \lhd s,$$

but this is impossible because $s \in \mathit{total}(M')$. Therefore $S = T$ and no such $p, q \in P$ exist.

Define $\mathsf{index}'$ of a view change message $m$ to be the unique value of $\mathsf{chnum}$ over all $\textcircled{p} \lhd m$ events; clearly $\mathsf{index}'$ is injective and the application of $\mathsf{index}'$ to the sequence of view change messages delivered by $p \in P'$ is a series of consecutive, increasing integers. Now let $\mathsf{index}$ be the restriction of $\mathsf{index}'$ to the view change messages of $\mathcal{C}$: because $\mathsf{index}'$ is injective, so is $\mathsf{index}$. Also, $\mathcal{C}$ is a subcollection of $\mathcal{C}'$, so the event sequences of $\mathcal{C}'$ are those of $\mathcal{C}$ with, perhaps, prefixes and suffixes, and the application of $\mathsf{index}$ to any sequence of view change messages delivered by $p \in P$ must be a series of consecutive, increasing integers.

Choose $p, q \in P'$ and let $m \in M'$ be a view change message. Let $S$ and $T$ be the set of all view change messages delivered before $m$ in $p$ and $q$'s event sequences, respectively. $S$ and $T$ are well-defined because, by the definition of communication collection, Ⓟ $\triangleleft m$ and Ⓠ $\triangleleft m$ occur only once in the sequences of $\mathcal{C}'$. We have already proven that

$$\mathsf{chnum}(Ⓟ \triangleleft m) = \mathsf{chnum}(Ⓠ \triangleleft m)$$

*i.e.*, that $|S| = |T|$, but suppose $S \neq T$. Then there must be view change messages $s \in S$ and $t \in T$ such that $s \notin T$ and $t \notin S$. Because $\mathcal{C}'$ is delivery atomic,

$$(Ⓟ \triangleleft s) \rightarrow_{\mathcal{C}'} (Ⓟ \triangleleft m) \rightarrow_{\mathcal{C}} (Ⓟ \triangleleft t) \;\text{ and }\; (Ⓠ \triangleleft t) \rightarrow_{\mathcal{C}'} (Ⓠ \triangleleft m) \rightarrow_{\mathcal{C}'} (Ⓠ \triangleleft s)$$

but this is impossible because $s, t \in \mathit{total}(M)$. Hence $S = T$ and, by the definition of $\lessdot$, $\lessdot_{\mathcal{C}'}(Ⓟ \triangleleft m) = \lessdot_{\mathcal{C}'}(Ⓠ \triangleleft m)$. Because $\mathcal{C}$ is a subcollection of $\mathcal{C}'$, $\lessdot_{\mathcal{C}}(Ⓟ \triangleleft m) = \lessdot_{\mathcal{C}}(Ⓠ \triangleleft m)$ for all $p, q \in P$ and view change messages $m \in M$. Therefore $\mathcal{C}$ is indexable. □

### 6.7.2 View Numbers

Given a communication collection $\mathcal{C}$, we say that a map $\mathsf{viewnum}$ from $\mathcal{C}$'s delivery events to the integers is a *view number generator* of $\mathcal{C}$ if:

1. there is a function $\mathsf{index}$ which indexes the views of $\mathcal{C}$, and

2. for each of $\mathcal{C}$'s delivery events that follows the delivery of a view change message, $\mathsf{viewnum}(e) = \mathsf{index}(m)$, where $m$ is the last view change message delivered before $e$ in $e$'s event sequence.

**Theorem 2.** If $\mathcal{C} = (P, M, \mathsf{seq}, \mathsf{Init})$ is virtually synchronous, then, for all $m \in M$ and $p, q \in P$,

$$[Ex(Ⓟ \triangleleft m) \wedge Ex(Ⓠ \triangleleft m)] \Rightarrow [\mathsf{viewnum}(Ⓠ \triangleleft m) = \mathsf{viewnum}(Ⓟ \triangleleft m)].$$

for some view number generator $\mathsf{viewnum}$ of $\mathcal{C}$.

**Proof.** Let $\mathcal{C}' = (P', M', \mathsf{seq}', \mathsf{Init}')$ be a supercollection of $\mathcal{C}$ as in the definition of virtual synchrony, and choose $m \in M'$, $p \in P'$ such that Ⓟ $\triangleleft m$ is an event in some sequence of $\mathcal{C}'$. Define the $\mathsf{chnum}$ and $\mathsf{index}'$ functions as in Theorem 1, *i.e.*, for all view change delivery events $e$ of $\mathcal{C}'$,

$$\mathsf{chnum}(e) = |\{e' \to e \mid e' \text{ is a view change delivery event}\}|,$$

and $\mathsf{index}'(m)$, for $m$ a view change message, is the unique value of $\mathsf{chnum}$ over all delivery events of $m$. Define the function

$$\mathsf{viewnum}'(ⓟ \triangleleft m) = \begin{cases} \mathsf{index}'(n) & \text{where } n \text{ is the last view change message} \\ & \text{delivered before } m \text{ in } p\text{'s sequence} \\ 0 & \text{if no such } n \text{ exists} \end{cases}$$

By definition, $\mathsf{viewnum}'$ is a view number generator for $\mathcal{C}'$.

Consider the restrictions $\mathsf{index}$ of $\mathsf{index}'$ to the view change messages of $\mathcal{C}$ and $\mathsf{viewnum}$ of $\mathsf{viewnum}'$ to the delivery events of $\mathcal{C}$; we saw in Theorem 1 that $\mathsf{index}$ indexes the views of $\mathcal{C}$. Let $ⓟ \triangleleft m$ be a delivery event of $\mathcal{C}$ such that there is a view change message $n$ for which

$$ⓟ \triangleleft n \to_{\mathcal{C}} ⓟ \triangleleft m.$$

Let $n'$ be the last such view change message. Then

$$\begin{aligned} \mathsf{viewnum}(m) &= \mathsf{viewnum}'(m) \\ &= \mathsf{index}'(n') \\ &= \mathsf{index}(n'). \end{aligned}$$

Thus $\mathsf{viewnum}$ is a view number generator for $\mathcal{C}$. Moreover, because each view change message $n \in M'$ is in $total(M')$,

$$\mathsf{viewnum}'(ⓟ \triangleleft m) = \mathsf{viewnum}'(ⓠ \triangleleft m)$$

for all $p, q \in P'$ and hence

$$[Ex(ⓟ \triangleleft m) \wedge Ex(ⓠ \triangleleft m)] \Rightarrow [\mathsf{viewnum}(ⓠ \triangleleft m) = \mathsf{viewnum}(ⓟ \triangleleft m)]$$

for all $m \in M$ and $p, q \in P$. $\square$

### 6.7.3 Stephenson's Definition

In [Ste91], Stephenson gives the following definition of virtual synchrony (using a different notation): A communication collection $\mathcal{C} = (P, M, \mathsf{seq}, \mathsf{Init})$ is *virtually synchronous* if, for all $p, q \in P$ and $m \in M$,

$$q \in \lessdot(ⓟ \lhd m) \Rightarrow Ex(ⓠ \lhd m) \wedge [\mathsf{viewnum}(ⓠ \lhd m) = \mathsf{viewnum}(ⓟ \lhd m)]$$

for some view number generator $\mathsf{viewnum}$ of $\mathcal{C}$.

**Theorem 3.** If $\mathcal{C}$ is virtually synchronous according to the definition of Section 6.7, then it is virtually synchronous according to Stephenson's definition.

**Proof.** Let $\mathcal{C}' = (P', M', \mathsf{seq}', \mathsf{Init}')$ be a supercollection of $\mathcal{C}$ as in the definition of virtual synchrony. Because $q \in \lessdot_{\mathcal{C}}(ⓟ \lhd m)$ and $\mathcal{C}$ is a subcollection of $\mathcal{C}'$, we know that $q \in \lessdot_{\mathcal{C}'}(ⓟ \lhd m)$. Moreover, from the definition of virtual synchrony, $\mathsf{Init}'(p) = \{\}$ for all $p \in P'$. Therefore, by the definition of $\lessdot$,

$$ⓟ \lhd join_q \rightarrow_{\mathcal{C}'} ⓟ \lhd m$$

and

$$Ex_{\mathcal{C}'}(ⓟ \lhd leave_q) \Rightarrow ⓟ \lhd m \rightarrow_{\mathcal{C}'} ⓟ \lhd leave_q.$$

Because $\mathcal{C}'$ is delivery atomic and $join_q, leave_q \in total_{\mathcal{C}}(M')$,

$$ⓠ \lhd join_q \rightarrow_{\mathcal{C}'} ⓠ \lhd m$$

and

$$Ex(ⓠ \lhd leave_q) \Rightarrow ⓠ \lhd m \rightarrow_{\mathcal{C}'} ⓠ \lhd leave_q.$$

By the definition of $\lessdot$, $q \in \lessdot_{\mathcal{C}'}(ⓠ \lhd m)$. Because $\mathcal{C}'$ is a non-presumptous supercollection of $\mathcal{C}$, $Ex_{\mathcal{C}}(ⓠ \lhd m)$. By the theorem above, then,

$$q \in \lessdot(ⓟ \lhd m) \Rightarrow Ex(ⓠ \lhd m) \wedge \mathsf{viewnum}(ⓠ \lhd m) = \mathsf{viewnum}(ⓟ \lhd m)$$

for some view number generator $\mathsf{viewnum}$ of $\mathcal{C}$. $\square$

Our definition of virtual synchrony and Stephenson's are not equivalent. In particular, Stephenson's definition does not have the property specified by Theorem 2. To see this, consider the following communication collection:

$$\mathcal{C} : \begin{cases} r: & \{\} \quad \textcircled{r} \rhd join_r,\ \textcircled{r} \lhd join_r,\ \textcircled{r} \rhd n,\ \textcircled{r} \lhd n,\ \textcircled{r} \rhd leave_r\ \textcircled{r} \lhd leave_r \\ s: & \{\} \quad \textcircled{s} \rhd join_s,\ \textcircled{q} \lhd join_s,\ \textcircled{q} \lhd n \end{cases}$$

This collection could describe an execution in which $r$ joins a group, sends $n$, and leaves the group; later, $s$ joins the group and receives $n$ (which was, perhaps, stuck in a buffer). Define index and viewnum by by

$$\begin{aligned} \mathsf{index}(join_r) &= \mathsf{viewnum}(\textcircled{r} \lhd join_r) = 1 \\ \mathsf{index}(leave_r) &= \mathsf{viewnum}(\textcircled{r} \lhd leave_r) = 2 \\ \mathsf{index}(join_s) &= \mathsf{viewnum}(\textcircled{s} \lhd join_s) = 3 \end{aligned}$$

It is not hard to show that index indexes the views of $\mathcal{C}$ and viewnum is the corresponding view number generator of $\mathcal{C}$. In fact, for all $p, q \in \{r, s\}$ and $m \in M = \{n\}$,

$$q \in \lessdot(\textcircled{p} \lhd m) \Rightarrow Ex(\textcircled{q} \lhd m) \wedge [\mathsf{viewnum}(\textcircled{q} \lhd m) = \mathsf{viewnum}(\textcircled{p} \lhd m)].$$

Therefore $\mathcal{C}$ is virtually synchronous according to Stephenson's definition.

Now let viewnum be any view number generator of $C$, and let index be its corresponding view indexer. By definition,

$$\mathsf{viewnum}(\textcircled{r} \lhd n) = \mathsf{index}(join_r)$$

and

$$\mathsf{viewnum}(\textcircled{s} \lhd n) = \mathsf{index}(join_s).$$

Because index is injective,

$$\mathsf{viewnum}(\textcircled{r} \lhd n) \neq \mathsf{viewnum}(\textcircled{s} \lhd n)$$

By the contrapositive of Theorem 2, $C$ is not really virtually synchronous according to the definition given in Section 6.7.

We believe that the absense of Theorem 2's property in Stephenson's definition is merely an oversight. Indeed, the protocol given by him in [Ste91] implements it, and it is implied by the informal specification[5] given by Birman, et al. in [BSS91].

[5] "All the recipients are in identical group views when the message arrives."

# Chapter 7

# Implementation

Large programming systems are usually subdivided into smaller units that can be programmed independently. Each unit, called a module, has an interface that other modules use to invoke its functionality. A typical interface includes a set of types, some external functions, and an (often informal) operational semantics of how the functions behave when invoked. In SML, a module is implemented with a structure, and its external types and functions are specified with a signature (described in Section 2.8). DML's current runtime system implementation is divided into three modules, one for intercolony communication, another for colony group support, and a third for port group support. This chapter describes the three module interfaces and explains how they work together to support DML's style of distributed programming.

## 7.1 Technical Summary

In practice, we would like DML's code modules to be as small as possible; small components are easier to reason about and to implement correctly than large ones. DML provides a number of properties that we would like to implement within independent code sections, including the implementations of unreliable and reliable data transmission, multicast atomicity, group and system-wide membership (including failure detection), data stability, and ordered events.

Unfortunately, interactions between components in a fault-tolerant communication system like DML are numerous (the implementors of Consul discovered this as well [HS93]). For instance:

1. Many components are directly affected by the actions of others. The membership module, for instance, affects ordering: data from a new port must be cleanly ordered into the existing data stream. The ordering and membership modules rely on the atomicity and reliable communication modules to implement their protocols. The atomicity module relies on the system membership module to eventually remove failed components.

2. Some system-wide properites depend on the overall behaviour of all components. For instance, liveness can be violated if one module blocks data that another needs and vice versa, and the correctness of an ordering module is meaningless if data is improperly rearranged after it leaves the module. Stability information is irrelevant if components can drop messages from the data stream.

The design of DML's runtime system required many hours of reasoning about interactions like these, and the existing solution is both modular and correct. It has evolved from a seemingly random set of active objects that communicate in arbitrary ways to the layered system presented in this chapter. Some properties that were enforced in all components—*e.g.*, "only ordering modules can block and reorder data," and "only the system membership module can drop data"—helped guarantee the system-wide properties. The implementation is similar to that of Horus [vRBGR93], even though the two systems were developed concurrently, but quite different from that of Isis [BJ87], which is only slightly modular. The DML runtime system is more

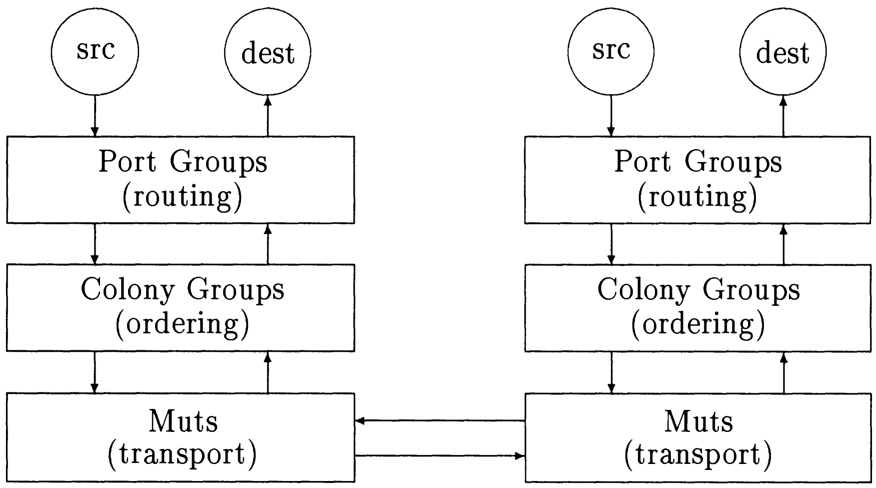



Figure 7.1: DML's communication layers

than 10,000 lines of CML code[1].

## 7.2 Implementation Overview

When DML data enters a source port, it is packaged into a message, tagged with ordering and delivery information, transmitted between colonies, ordered relative to other data in the DML system, and delivered to the appropriate destination ports. The method used to accomplish these tasks resembles an automatic car wash: as a car moves along a conveyer, it is soaped, scrubbed, rinsed, waxed, and dried by separate machines that work in a fixed position. Similarly, DML data passes through *layers* of the runtime system that package, tag, transmit, order, and deliver it. The layers are called the *Multicast Transport* (or MUTS) layer, the *Colony Group* layer, and the *Port Group* layer.

The data path in a two-colony DML system is illustrated in figure 7.1.

[1]Although it is difficult to compare the compactness of programming languages, this is comparable to 30,000 lines of C code.

Once data has entered a source group, it is passed to an instance—determined by the source port's colony—of the port group layer. This layer packages the data into a message, tags it with routing information, and passes it to an instance of the colony group layer. The colony group layer adds ordering information to the message and forwards it to an instance of the MUTS layer, which transmits it to other MUTS layers in the system using mechanisms (*e.g.*, shared memory, datagrams, streams, and hardware multicast) provided by the operating system. A remote instance of MUTS then passes the data to its corresponding colony group layer which decodes ordering information and arranges the data relative to other incoming messages. The ordered data is then passed to the port groups layer and is routed to one or more destination port. Each three-component layer stack is specific to a port colony; because port colonies share heaps and machines, there can be multiple layer stack instances within a single address space. The layer instances are composed of active objects and are implemented as discussed in Section 3.3.

A simple way to implement routing in DML would require each port colony to distribute its outgoing data to all MUTS instances in the system. Data that is not needed at a colony (because it does not contain an appropriate destination port) would be dropped when it reached the colony's port group layer. This scheme has several advantages: it is easy to describe, its correctness properties are relatively simple, and its ordering and membership protocols are not complex (largely because all transmitted information is available everywhere). Unfortunately, it is not scalable; protocols used by the system employ $O(n)$ messages, where $n$ is the number of colonies in the system.

A better approach would only distribute outgoing data to colonies that contain a destination port of the appropriate port group; this would require $O(g)$ messages, where $g \leq n$ is the number of colonies that require the information. For this purpose, the colony group layers maintain a set of *colony groups* that can be destinations of outgoing data. Colony groups are not CML objects like ports and layers; instead, they are internal constructs: each member of the group maintains information chunks that together constitute the colony group. Colonies join and leave colony groups by communicating with the group's members and updating internal information. DML's ordering protocols only work within colony groups and cannot order information between them[2]. In fact, each colony group corresponds to an ordering preserve; when two port groups share a preserve, they use the same colony group to distribute their data through the system. A system like this is scalable to an arbitrary number of colonies as long as the size of colony groups—*i.e.*, ordering preserves—remain relatively small.

Ordered events are implemented by the destination ports. Each colony group layer constructs a totally-ordered stream of data consistent with the partial order required by the ordering constraints. When the `decide` operator (see Section 4.6) is applied to ordered delivery events, the destination ports are polled and the earliest data item in the colony group layer's total order that is destined for one of the ports—if any are available—is returned.

In the rest of this chapter we discuss the properties and guarantees provided by each layer of DML's runtime system. We also describe how each layer is implemented and observe how the modules interact.

## 7.3 MUTS

The Multicast Transport System (MUTS) transmits messages between colonies. It provides a transmissable datatype, a reliable but nonatomic multicast to lists of destinations, a means of monitoring unresponsive destinations, and a primitive name service for bootstrapping. DML's MUTS layer is patterned after Robbert Van Re-

[2]At one time DML supported *multigroup causality*, but it became unclear that this feature was worth the extra implementation and maintenance effort. It could be added again.

Table 7.1: MUTS Data Objects

| **Data object** | **Description** |
|---|---|
| Connection | The state needed to implement the lowest layer of a colony. DML opens a connection whenever a colony is created. |
| Endpoint ID | A portable description (or address) of a connection. An endpoint ID may contain the host machine, Unix process number, a socket descriptor and any other information necessary to communicate with its connection. |
| Group ID | A record used to uniquely identify a group of connections. A group ID does not contain member-specific information. |
| Handle | The state needed to transmit a stream of messages to a set of connections. Handles are bound to a connection and may, *e.g.*, refer to the local state of a sliding window protocol. |
| Message | Transmittable data supplied by layers above MUTS. |

nesse's system of the same name, which is written in C [vRBC+92].

### 7.3.1 Specification

Table 7.1 summarizes the principle data objects used by MUTS to implement its services. A MUTS *connection* encompasses the per-colony state necessary to send and receive messages. When a DML colony is created, the runtime system allocates a MUTS connection to support its communication needs and creates an *endpoint ID* that describes the new colony. Endpoint ID's are portable—they can be inserted into messages—and contain enough information to initiate communication with a remote connection. As an endpoint ID describes a single connection, a *group ID* describes a set of connections; because group ID's do not contain member-specific information, they can be used to identify groups with dynamic membership. A *handle* is the state required to transmit messages from one connection to a group of connections. Handles are bound to both a connection and a group ID and may, *e.g.*, refer to the local state of a sliding window protocol. Messages are application data that can be communicated between connections.

The MUTS layer supports communication between connections by providing four operations—register, add/delete, send, and monitor—to DML's other layers. When a colony $C$ joins a colony group $G$, it *registers* for data addressed to $G$. After registration, $C$'s connection will deliver all locally-received data marked by $G$'s group ID. Registering for group data is not sufficient to receive all $G$-addressed data: members of $G$ must be made explicitly aware of the joining colony before they send data to it. Membership coordination is handled by DML's colony group layer.

A colony can *add* or *delete* a connection from a group by presenting an endpoint ID to a handle. A handle maintains a current endpoint ID list which defines its destination set, and a colony *sends* a message to that set by specifying a message to the handle. The handle tags the message $m$ with the handle's group ID $G$ and transmits $m$ to each connection specified by its endpoint ID list. If a receiving connection has an outstanding registration for $G$, $m$ is delivered to the appropriate colony group layer. Messages are delivered in *fifo* order; that is, if message $m_1$ is sent on a handle before $m_2$, $m_1$ will arrive before $m_2$ at all destinations.

Sometimes a handle cannot complete a send because one of its destinations has stopped, crashed, or disconnected from the network. A colony must *monitor* connections to detect that something has gone wrong. A handle will not stop sending to a failed connection unless the errant endpoint ID is deleted by one of DML's other layers.

MUTS also provides a system-wide name service to bootstrap a DML session. The name service is similar to a distributed file system, and it is neither efficient nor fault-tolerant.

### 7.3.2 Implementation

Three separate implementations of DML's MUTS layer exist, but only one of them has been extensively tested. The *MutSim* implementation simulates an network within a single Unix process, and can be used to create multiple colonies within a single heap. Although MutSim cannot be used for true distributed programming, it works correctly, and, because latencies of the simulated network can be modified dynamically, MutSim is useful for detecting rare race conditions in application code.

*Uros* is a MUTS implementation that works by communicating with one of Robbert van Renesse's MUTS processes (written in C) through a Unix-domain socket connection. With Uros, each MUTS operation invoked by DML is relayed to the C process, and each message received by the C process is forwarded to Uros. Although inherently slow, Uros can be used for true distributed programming. Uros is currently in a testing phase and has not passed any DML stress tests.

The *MutSML* implementation of MUTS is designed to directly support distributed programming from within a DML heap. It uses common networking protocols like UDP and TCP, and supports its own sliding window protocols. Unfortunately, MutSml does not implement the complete MUTS specification and its author does not intend to upgrade it. MutSim was written by the author of this dissertation; Uros is by Robert Cooper, and MutSML was developed by Jaideep Vijan.

## 7.4 Colony Groups

DML's Colony Group layer is responsible for ordering messages, providing stability information, ensuring multicast atomicity, and maintaining the colony groups. It is divided into four distinct sublayers (*i.e.*, objects): the *failure detector*, the *message stabilizer*, the *membership bureau*, and the *protocol disk*. These layers are organized as illustrated in Figure 7.2. The failure detector provides system membership infor-

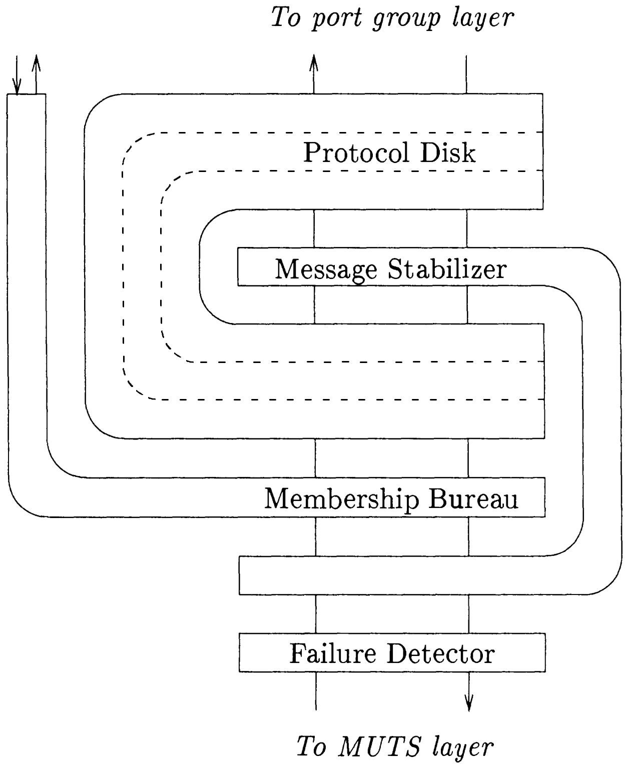



Figure 7.2: The colony group layer

mation to the other components of the system, and the message stabilizer implements a multicast atomicity protocol by storing received messages until they become stable. The membership bureau informs the other components when other colonies join and leave colony groups; it guarantees that once it declares that a colony $C$ has left a group $G$, no more messages from $C$ to $G$ exist between its two arms (this explains its odd shape—see Figure 7.2). As illustrated by the two short arrows in Figure 7.2, the bureau also provides membership information to the port group layer. The ordering disk implements group-based causal, total, and universal orderings, and it only blocks and reorders data within its upper arm (see Figure 7.2). This property prevents deadlock between the membership bureau and the ordering disk, as the membership bureau flushes the data stream between its two arms before announcing that a colony has left a colony group.

A message $m$ flows from the port group layer to the protocol disk. The protocol disk, responsible for ordering messages before delivery, tags $m$ with ordering information and hands $m$ to the message stabilizer. The stabilizer works to ensure that $m$ and other messages are received at all intended destinations that do not fail. After examining and tagging $m$, the stabilizer passes the message back to the protocol disk, which forwards $m$ to the membership bureau. The bureau is the local object responsible for maintaining the current membership lists—or *views*—of known colony groups. The bureau passes $m$ to the failure detector, which sends it to MUTS. The failure detector maintains the list of active colonies in the system; this list is the same at every failure detector object in the system and is used to make uniform decisions about dynamic changes to the DML environment.

After a message $m$ has been received by MUTS, it is sent to the failure detector. The failure detector decides if $m$'s sender is a valid colony; if it is, it passes $m$ to the message stabilizer, and otherwise it drops it. The message stabilizer stores $m$ until $m$ has become completely stable—*i.e.*, until the local stabilizer knows that $m$ has been received at all of its destinations—and passes $m$ to the membership bureau. Should $m$'s sender fail, the stabilizer forwards a copy of it to all of $m$'s receivers, ensuring that the message is received by all or none of its intended destinations. The membership bureau passes $m$ to the ordering disk, which decodes the ordering information stored with the messsage and, if necessary, stores the message in a queue until it can be safely delivered. Once all of $m$'s predecessors have been released by the disk, $m$ is delivered to DML's port group layer. The next four sections examine the implementation of each colony group layer component in detail.

### 7.4.1 Failure Detector

The failure detector maintains a list of colonies that are not eligible to participate in distributed conversations. Colonies can be removed from the system membership for several reasons; for instance, the application that created them may raise an uncaught exception or exhaust virtual memory, a user may kill the heap containing the colony (*e.g.*, with Unix's `kill(1)`), or the colony's machine may crash or become disconnected from the network. In systems where communication time is unbounded, communication participants cannot reliably detect crashes; failures in such systems are indistinguishable from unresponsive machines [FLP85]. DML's failure detector removes colonies which appear to be dead and then drops messages from *zombies*—colonies that were classified as dead yet live—should any appear.

DML's current failure detector implementation, described below, is simple but not fault tolerant; the entire DML system will stop if the oldest colony in the system dies. A more complex but completely fault tolerant failure detector is given in [Ric92] and should eventually replace the initial implementation.

When a failure detector object is created, it contacts the name service provided by MUTS and tries to install itself as the first colony in the system. If this succeeds, the querying object is the *coordinator* of the failure detection service; if the operation fails (because the coordinator had already registered itself), it contacts the existing coordinator and asks for the current list of deceased colonies. If it cannot contact the coordinator, the failure detector object's create operation fails. Otherwise, the coordinator transfers the list as requested and the local failure detector and its colony formally join the DML system.

The failure detector uses the monitoring facility of MUTS (see Section 7.3.1) to determine when other colonies are being unresponsive[3]. When a local detector object suspects that a colony has failed, it sends a suspicion notice to the failure detector coordinator. The coordinator confirms the death of the suspected colony by telling each colony in the system to update their list of failed colonies appropriately. Occasionally all colonies send null ("keep alive") messages to the coordinator, and if the coordinator receives a message from a zombie, it tells the zombie that it has been removed from the system. If a colony cannot talk to the coordinator, it removes itself from the membership.

An important property of this implementation (and the fault tolerant one given in [Ric92]) is this: once a colony $C$ has announced the removal of a colony $C'$, then all other colonies in the system either crash, kill themselves, or remove $C'$ from the system. The failure detector also guarantees that DML's upper layers will never see a message from a zombie colony.

### 7.4.2 Message Stabilizer

A message is *fully stable* when it has been delivered to the message stabilizer object at each of its destinations. The stabilizers maintain tables of unstable messages so that, if a sender fails and causes a message $m$ to be delivered to a proper subset of its destinations, $m$ can be retransmitted. The stability information is attached to the

[3]DML applications can also cause a suspicion by calling the function `suspect` on a `port_id`. This can help during the test phase of a fault tolerant program.

messages as they pass through the stabilizers, so other DML layers can use stability information for their own purposes.

Like the colony group layer, the message stabilizer is composed of multiple active objects. A single *router* object routes messages between multiple *clerks*, where there is one clerk per colony group that contains the local colony. Each clerk has a message table in which it stores unstable messages. When a message $m$ is received by a message stabilizer, the appropriate clerk sends a reply to $m$'s sender. When every destination has acknowledged $m$, the sending stabilizer clerk determines that $m$ is stable and forwards this information to the destination stabilizers (often by "piggybacking" the stability information to other outgoing messages). When a receiving stabilizer clerk learns that $m$ is fully stable, it deletes $m$ from its message table.

When a failure detector object declares that a colony $C$ has failed, the stabilizer clerks initiate a *fail flush* protocol for $C$. Every clerk forwards its unstable messages from $C$ to the oldest member of the colony group which, after receiving a message from all other clerks, forwards its unstable messages (including those received during the fail flush) to the members. Once a fail flush has completed, all messages from $C$ are fully stable and can be removed from the clerks' message tables.

If the oldest member of a colony group fails during a fail flush, all current flushes are cancelled and a new one is initiated. To ensure that the protocols terminate, colonies are not allowed to join groups that have ongoing fail flushes. Before a stabilizer clerk announces that a fail flush has completed for a colony $C$, it sends a "sweep $C$" message from the upper arm of the stabilizer (see Figure 7.2) to the lower arm and vice versa. The sweep ensures that no message from $C$ remains between the stabilizer arms; when the fail flush of $C$ is announced, other layers of the system can know that they will not see a message from $C$ in that part of the system again.

### 7.4.3 Membership Bureau

Colonies join and leave colony groups by requesting services from the membership bureau. If a port $p$ is a member of a colony $C$ and a port group $G$, $C$ must belong to the colony group that supports $G$'s communication. A port create operation may cause a colony to join the appropriate colony group, and a port remove may allow a colony to leave a group.

The membership bureau consists of four types of active objects: *routers*, *joiners*, *leavers*, and *autojoiners*. Each bureau harbors one router, a joiner and leaver per colony group containing the local colony, and an autojoiner per uncompleted join attempt operation. The router handles all incoming requests—messages, join and leave requests, and failure notifications—and routes them to the appropriate object; it also notifies the other layers when a colony group membership change occurs. The joiner and leaver objects implement the join and leave protocols, respectively. The autojoiner object participates in the initial part of the join protocol before a colony has successfully joined a colony group.

The port group layer requests a join of a colony group $G$ by presenting the group's MUTS group ID and a list of the group's members $M$ to the router. The router spawns an autojoiner, and the autojoiner asks $C$, the oldest member of $M$, to sponsor its colony in a join operation by sending it a JOIN REQUEST message. If $C$ refuses refuses (because it has left $M$) or fails, the autojoiner sends a JOIN REQUEST to the next oldest member of $M$, and repeats. If every member of $M$ either refuses—by sending a JOIN REFUSED—or crashes, the join operation fails, and the port group layer is told that the set $M$ is *stale* (this usually causes a port create operation to fail). If some member of $G$ agrees to support the join, it confers with the members of $G$ (see below) and returns a JOIN SUCCESS message containing the current membership. The autojoiner sends a JOIN COMMIT message to $G$ and passes the membership to the router. The router constructs a joiner and a leaver

Table 7.2: Messages of the join protocol

| Message | Sender | Receiver |
|---|---|---|
| JOIN REQUEST | joining member | oldest member |
| JOIN ATTEMPT | oldest member | all members |
| JOIN OK | all members | oldest member |
| JOIN SUCCESS | oldest member | joining member |
| JOIN COMMIT | new member | all members |
| JOIN REFUSED | non-member | joining member |
| JOIN ABORT | ex-joining member | all members |

for $G$, informs the colony about the new group, and returns success to the port group layer. If a router receives a JOIN SUCCESS message of a group $G$ with no corresponding autojoiner, it reasons that an autojoiner had quit prematurely, and sends a JOIN ABORT message to the members of $G$ encoded in the message. The messages sent in the join protocol are summarized in Table 7.2. A colony has one joiner for every group to which it belongs; the joiner handles join protocol messages. If it receives a JOIN REQUEST from a colony, it multicasts a JOIN ATTEMPT to all members of the group and waits for each to reply with a JOIN OK. It then sends a JOIN SUCCESS to the joiner and waits for a reply, either JOIN COMMIT or JOIN ABORT. Only one join is allowed to be active at a time, and joins are serialized by the oldest member of the group.

The leave protocol is similar to the join protocol; its messages are summarized in Table 7.3. When a router receives a leave request from the port group layer, it forwards the request to the appropriate leaver, and the leaver sends a LEAVE REQUEST to $C$, the oldest member of the group. $C$'s leaver multicasts a LEAVE ATTEMPT to all members of the group. Each member prepares itself for the imminent departure and acknowledges $C$ with a LEAVE OK message. After receiving an acknowledgment from each member, $C$ sends LEAVE SUCCESS to the leaving colony. The leaving member then multicasts LEAVE COMMIT to all members and reports success to the

Table 7.3: Messages of the leave protocol

| Message | Sender | Receiver |
|---|---|---|
| LEAVE REQUEST | leaving member | oldest member |
| LEAVE ATTEMPT | oldest member | all members |
| LEAVE OK | all members | oldest member |
| LEAVE SUCCESS | oldest member | leaving member |
| LEAVE COMMIT | leaving member | all members |

port group layer. If $C$ fails during the protocol, the leaving colony restarts the protocol by sending a LEAVE REQUEST to the next oldest member. The oldest member of the group does not initiate a leave protocol unless there are no active joins.

### 7.4.4 Protocol Disk

The protocol disk enforcing message ordering constraints; it is the only layer in DML that can block and reorder message streams. In fact, it only blocks messages within its upper arm (see Figure 7.2), and other colony group objects rely on this property for liveness. The protocol disk is divided into three sublayers, the *universal belt*, the *total belt*, and the *causal belt.* These layers are arranged as illustrated in Figure 7.3.

If a message $m$ is *universally ordered*, then, for all other messages $n$ sent to the same colony group, $m$ and $n$ must be delivered in the same order at any colony that delivers them both. The universal belt enforces this ordering property by delaying a universally ordered message $m$ until a fault-tolerant *cut flush* protocol completes. The flush ensures that each group member delivers all messages sent before $m$ or concurrent with $m$. The protocol is similar to the fail flush protocol described in Section 7.4.2 and, like that protocol, disables joins before attempting to flush.

If $m$ and $n$ are *totally ordered*, they must be delivered in the same order at any colony that delivers them both. The total belt delays totally ordered messages until their ordering is set by a special member of the group called the *token holder.* The

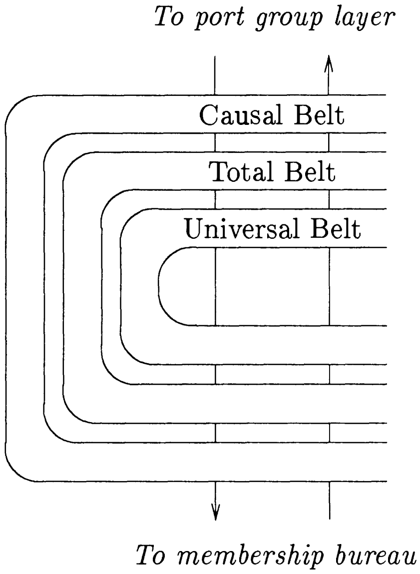



Figure 7.3: Components of the protocol disk

token holder need not delay totally-ordered messages; instead, it delivers them as it receives them and sends a SETS ORDER message to other members of the group. The total belt only allows one totally-ordered message at a time to exist between it and the port group layer.

Causally ordered messages are reordered within the causal belt, which uses the single group vector timestamp algorithm described in [Ste91]. Because the total belt only issues one totally-ordered message at a time, the causal belt need not worry about incorrectly rearranging totally-ordered messages.

## 7.5 Port Groups

DML's port group layer routes messages between ports and the colony group layer and implements ordered events. When an instance of the layer is asked to create a port of the port group $G$, it determines if its colony is currently a member of $G$'s colony group and, if not, requests a colony group join from the membership bureau of the colony group layer. Once it joins $G$'s colony group, the instance sends a join notification to the meta ports of $G$ and returns a new port to the application.

When an application removes the port $p$ from the system, $p$'s colony group layer sends a leave notification to the meta ports of $p$'s port group and, if it is not responsible for any other ports corresponding to $p$'s ordering preserve, asks its membership bureau to leave the colony group. The port group layer also receives incoming data from the colony group layer and routes it to the appropriate destination ports.

If a thread applies the `decide` function (see Section 4.6) to a list of ordered events from destination ports that share an ordering preserve[4], the port colony layer corresponding to the ordering preserve temporarily stops receiving messages from its colony group layer and searches for the least-ordered data item available at one of the specified destination ports. If nothing appropriate is found, it queues the thread and resumes processing its incoming data stream. Each message from the colony group layer is examined to see if it can be delivered to one of the queued threads.

[4]Applying `decide` to ordered events from destination ports in different ordering preserves raises an exception; this capability seems unnecessary and complicates the implementation.

# Chapter 8

# Conclusions

In this thesis, we discussed the design and implementation of a new distributed programming languaged called Distributed ML (DML). Because DML's communication primitives are asynchronous, the language can be used to program efficient applications that execute over loosely-coupled distributed systems, like sets of workstations connected through a relatively low-speed network. DML adds an asynchronous group-based communication object to the Concurrent ML language and provides ordered event objects for constructing abstract communication patterns. DML is the first programming language to provide an asynchronous, fault-informative, multicast object, and the use of ordered events is DML's unique and powerful way of encapsulating ordering information within application programs. As we demonstrated in Chapter 5, DML's primitives are both powerful and general, and DML programmers can write simple programs to solve a wide variety of difficult distributed problems.

In Chapter 6, we introduced a formal notation for reasoning about the possible communication histories in DML programs and show that, using a very natural restriction on the order of delivery events, the run time system enforces the *virtual synchrony* communication model made popular by the Isis Distributed Toolkit [BJ87, BCG91]. We believe that ours is the first definition of virtual synchrony that captures its intuitive meaning; previous formal definitions are combinations of message existence and ordering constraints that fail to convey its purpose or usefulness.

Implementing a fault tolerant distributed system is difficult; resilient protocols are usually complex [NT88], and systems are difficult to modularize [HS93]. In Chapter 7, we described the implementation of DML's primitives and discussed how the implementation was divided into semi-independent modules. We believe that the DML architecture is an excellent way to subdivide a system with objects that communication in complex and failure-prone ways. DML's MUTSsimulator is a good tool for testing DML programs before they are migrated to a distributed system.

In retrospect, we would have designed DML differently if we were to do so with two years of implementation and coding experience. Although we still believe that DML's explicit failure information and asynchronous primitives are vital to any general-purpose distributed languages, we would not have designed the core language with an atomic and ordered group-based primitive. Both multicast atomicity and ordering primitive impose overhead on message transmissions that unduly impedes unordered failure-free communication, and we are not convinced that atomicity and order are useful as a general-purpose communication guarantees. We would still have provided a weaker group-based primitive so that DML could support hardware multicast directly, but the more expensive multicast properties would have been implemented in a library on top of the core language. This design is not fixed, but SML's module system would make it possible to integrate non-primitive communication mechanisms into programs designed to use DML's primitives.

The lack of a dynamic type or some other method of implicitly attaching marshalling functions to SML types hampered much of DML's interface development and complicated its signature. Although DML was originally intended to support dynamic types, the necessary work never materialized and group type objects have proliferated and propogated through its implementation and coding examples.

# Bibliography


[ABG$^+$91] J. S. Auerbach, D. F. Bacon, A. P. Goldberg, G. S. Goldszmidt, M. T. Kennedy, A. R. Lowry, J. R. Russell, W. Silverman, R. E. Strom, D. M. Yellin, and S. A. Yemini. High-level lanugage support for programming reliable distributed systems. Technical Report RC 16441, IBM T. J. Watson Research Center, January, 1991.

[AD76] P. A. Alsberg and J. D. Day. A principle for resilient sharing of distributed resources. In *Proceedings of the Second International Conference on Software Engineering*, pages 627–644, October 1976.

[ADKM92] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: A communication sub-system for high availability. In *Proceedings of the 22nd Annual International Symposium on Fault-Tolerant Computing*, pages 76–84, July 1992.

[AHJ91] Mustaque Ahamad, Phillip W. Hutto, and Ranjit John. Implementing and programming causal distributed shared memory. In *Proceedings of the 11th International Conference on Distributed Computing*, pages 274–281, May 1991.

[AM87] A. W. Appel and D. B. MacQueen. *A Standard ML Compiler*, volume 274, pages 301–324. Springer-Verlag, September 1987.

[AM91] A. W. Appel and D. B. MacQueen. Standard ML of New Jersey. In *Programming Language Implementation and Logic Programming*, volume 528 of *Lecture Notes in Computer Science*, pages 1–26. Springer-Verlag, August 1991.

[AO85] Gregory R. Andrews and Ronald A. Olsson. Report on the distributed programming language SR. Technical Report 85-23, Department of Computer Science, The University of Arizona, November 1985.

[App92] Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press, New York, NY, 1992.

[Bar84] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, New York, NY, 1984.

[BBG+89] A. Borg, W. Blau, W. Gretsch, F. Herrmann, and W. Oberle. Fault tolerance under Unix. *ACM Transactions on Computer Systems*, 7(1):1–23, February 1989.

[BC90] Kenneth Birman and Robert Cooper. The Isis project: Real experience with a fault tolerant programming system. Technical Report 90-1138, Department of Computer Science, Cornell University, July 1990.

[BCG91] Kenneth P. Birman, Robert Cooper, and Barry Gleeson. Design alternatives for process group membership and multicast. Technical Report 91-1257, Department of Computer Science, Cornell University, December 1991.

[Bir91] Kenneth P. Birman. Maintaining consistency in distributed systems. Technical Report 91-1240, Department of Computer Science, Cornell University, December 1991.

[BJ87] Kenneth P. Birman and Thomas A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, February 1987.

[Bla92] Uyless D. Black. *TCP/IP and related protocols*. McGraw-Hill, New York, NY, 1992.

[BN84] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[BSS91] Kenneth P. Birman, Andre Schiper, and Patrick Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, August 1991.

[BT88] Henri E. Bal and Andrew S. Tanenbaum. Distributed programming with shared data. In *Proceedings of the IEEE CS 1988 International Conference on Computer Languages*, pages 82–91, October 1988.

[BvR92] Kenneth P. Birman and Robbert van Renesse. RPC considered inadequate, 1992. In preparation.

[CLNW89] S. M. Clamen, L. D. Leigengood, S. M. Nettles, and J. M. Wing. Reliable distributed computing with avalon/common lisp. Technical Report CMU-CS-89-186, School of Computer Science, Carnegie-Mellon University, September 1989.

[Coo90] Eric C. Cooper. Programming language support for multicast communication in distributed systems. Technical Report 90-121, Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA, May 1990.

[DC90] S. E. Deering and D. R. Cheriton. Multicast routing in datagram internetworks and extended LANs. *ACM Transactions on Computer Systems*, 8(2), May 1990.

[Dep83] U.S. Department of Defense. *Reference manual for the Ada programming language.* ANSI/MIL-STD-1815A, DoD, Washington, DC, 1983.

[Ett87] D. M. Etter. *Structured FORTRAN 77 For Engineers and Scientists.* Benjamin/Cummings, Reading, MA, second edition, 1987.

[FLP85] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

[FW85] William Findlay and David A. Watt. *Pascal: an Introduction to Methodical Programming.* Houghton Mifflin, Boston, MA, third edition, 1985.

[Geh84] N. H. Gehani. Broadcasting sequential processes (BSP). *IEEE Trans. Softw. Eng.*, SE-10(4):343–351, July 1984.

[GLS90] Jr. Guy L. Steele. *Common Lisp: The Language, Second Edition.* Digital Press, 1990.

[GMP89] A. Giacalone, P. Mishra, and S. Prasad. Facile: a symmetric integration of concurrent and functional programming. *International Journal of Parallel Programming*, 18(2):121–160, 1989.

[GMP90] A. Giacalone, P. Mishra, and S. Prasad. Operational and algebraic semantics for Facile: A symmetric integration of concurrent and functional programming. In *Proceedings of the ICALP 90*, volume 443 of *Lecture Notes in Computer Science*, pages 765–780. Springer-Verlag, 1990.

[GMW79] Michael J. C. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation.* Springer-Verlag, 1979.

[HMV93] My Hoang, John Mitchell, and Ramesh Viswanathan. Standard ML weak polymorphism and imperative constructs. In *Proceedings of the*

*Eigth Annual Symposium on Logic in Computer Science*, page To appear, June 1993.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[HS93] Matti A. Hiltunen and Richard D. Schlichting. An approach to constructing modular fault-tolerant protocols. Technical Report TR 93-10, Department of Computer Science, The University of Arizona, March 1993.

[Hud86] P. Hudak. Para-functional programming. *Computer*, 19(8):60–70, August 1986.

[Hut87] N. C. Hutchinson. Emerald: An object-based language for istributed programming. Technical Report 87-01-01, Department of Computer Science, University of Washington, Seattle, WA, January 1987.

[Inm84] Inmos Ltd. *Occam Programming Manual.* Pretnice-Hall, Englewood Cliffs, NJ, 1984.

[Kna92] Frederick Knabe. A distributed protocol for channel-based communication with choice. Technical Report ECRC-92-16, European Computer-Industry Research Centre, Arabellastr. 17, W-8000 Munich 81, Germany, 1992.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language, Second Edition.* Prentice Hall, Englewood Cliffs, NJ, 1988.

[Kra92] André Kramer, January 92. personal communication.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[LC84] Thomas J. LeBlanc and Robert P. Cook. Boradcast communication in starmod. In *Proceedings of the 4th International Conference on Distributed Computing Systems*, pages 319–325, May 1984.

[LH86] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. In *Proceedings of the 5th Annual ACM Symposimu on Principles of Distributed Computing*, pages 229–239, August 1986.

[Lis88] Barbara Liskov. Distributed programming in Argus. *Communications of the ACM*, 31(3):300–312, March 1988.

[LLS90] Rivka Ladin, Barbara Liskov, and Liuba Shrira. Lazy replication: Exploiting the semantics of distributed services. In *Proceedings of the Tenth ACM Symposium on Principles of Distributed Computing*, pages 43–58, August 1990.

[LM91] Xavier Leroy and Michel Mauny. Dynamics in ML. In *Functional Programming Languages and Computer Architecture '87*, volume 523 of *Lecture Notes in Computer Science*, pages 406–426. Springer-Verlag, August 1991.

[MB76] R. M. Metcalfe and D. R. Boggs. Ethernet: Distributed packet switching for local computer networks. *Communications of the ACM*, 19(7):395–404, 1976.

[Min89] S. E. Minzer. Broadband ISDN and asynchronous transfer mode (ATM). *IEEE Commun. Mag.*, 27(9):17–24,57, September 1989.

[MPS92] Shivakant Mishra, Larry L. Peterson, and Richard D. Schlichting. Experience with modularity in Consul. Technical Report 92-25, Department of Computer Science, University of Arizona, Tuscon, AZ, August 1992.

[MT91] R. Milner and M. Tofte. *Commentary on Standard ML.* The MIT Press, Cambridge, Mass, 1991.

[MTH89] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, Cambridge, Massachusetts, 1989.

[Nel81] B. J. Nelson. Remote procedure call. Technical Report 81-119, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, May 1981.

[Nel91] Greg Nelson, editor. *Systems Programming with Modula-3.* Prentice Hall, Englewood Cliffs, NJ, 1991.

[NT88] Gil Neiger and Sam Toueg. Automatically increasing the fault-tolerance of distributed algorithms. In *Proceedings of the Seventh Symposium on Principles of Distributed Computing*, pages 248–262, August 1988.

[NW91] S. M. Nettles and J. M. Wing. Persistence + undoability = transactions. Technical Report CMU-CS-91-173, School of Computer Science, Carnegie-Mellon University, August 1991.

[Pau91] Laurence C. Paulson. *ML for the working programmer.* Cambridge University Press, Computer Laboratory, University of Cambridge, 1991.

[PBS89] Larry L. Peterson, Nick C. Bucholz, and Richard D. Schlichting. Preserving and using context information in interprocess communication. *ACM Transactions on Computer Systems*, 7(3):217–246, August 1989.

[Rea89] Chris Reade. *Elements of Functional Programming.* Addison-Wesley, New York, NY, 1989.

[Rep89] John H. Reppy. First-class synchronous operations in Standard ML. Technical Report 89-1068, Department of Computer Science, Cornell University, December 1989.

[Rep90] John H. Reppy. Concurrent programming with events: The Concurrent ML manual. Accompanies the most recent Concurrent ML release, November 1990.

[Rep91] John H. Reppy. CML: A higher-order concurrent language. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 293–305, June 1991.

[Rep92] John Hamilton Reppy. High-order concurrency. Technical Report 92-1285, Department of Computer Science, Cornell University, June 1992.

[Ric92] Aleta Marie Ricciardi. The group membership problem in asynchronous systems. Technical Report 92-1313, Department of Computer Science, Cornell University, Ithaca, NY, November 1992.

[Ros86] F. E. Ross. FDDI—a tutorial. *IEEE Commun. Mag.*, 24(5):10–17, May 1986.

[RT74] D. M. Ritchie and K. Thompson. The UNIX time-sharing system. *Communications of the ACM*, 17(7):365–375, July 1974.

[SG90] R. W. Scheifler and J. Gettys. The X window system. *ACM Transactions on Graphics*, 5(2):232–245, April 1990.

[SRC84] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, November 1984.

[Ste91] Patrick Stephenson. Fast ordered multicasts. Technical Report 91-1194, Department of Computer Science, Cornell University, Ithaca, NY, February 1991.

[TL93] Chandramohan A. Thekkath and Henry M. Levy. Limits to low-latency communication on high-speed networks. *ACM Transactions on Computer Systems*, 11(2):179–203, May 1993.

[Tof90] Mads Tofte. Type inference for polymorphic references. *Information and Computation*, 89:1–34, 1990.

[TvR88] A. S. Tanenbaum and R. van Renesse. A critique of the remote procedure call paradigm. In *Proceedings of the EUTECO '88 Conference*, pages 775–783, April 1988.

[vRBC+92] Robbert van Renesse, Kenneth Birman, Robert Cooper, Bradford Glade, and Patrick Stephenson. Reliable multicast between microkernels. In *Proceedings of the USENIX workshop on Micro-Kernels and Other Keernel Architectures*, April 1992.

[vRBGR93] Robbert van Renesse, Kenneth P. Birman, Brad Glade, and Mike Reiter. Design and performance of the Horus multicast system, 1993. In preparation.

[Wri93a] Andrew Wright, March 1993. Personal communication.

[Wri93b] Andrew K. Wright. Polymorphism for imperative languages without imperative types. Technical Report 93-200, Department of Computer Science, Cornell University, February 1993.