AUTOMATIC GENERATION OF COMMUNICATION PROTOCOLS

by

Bryan S. Rosenburg

Computer Sciences Technical Report #670
October 1986

AUTOMATIC GENERATION OF COMMUNICATION PROTOCOLS

by

BRYAN S. ROSENBURG

A thesis submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy
(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN — MADISON

1986

Abstract

This dissertation describes an effort to improve application-level communication efficiency by using information provided by, and derived from, the application itself.

We define a simple language for describing the static form of conversations between application processes. A conversation description serves as a form of service specification for the conversation's underlying communication protocol. It specifies the types of messages required by the conversation and provides a regular expression that defines the set of legal sequences of messages between the conversation's participants.

We also define structures we call plans that application processes can use dynamically to participate in conversations. A plan is a regular expression a process can construct to describe its desire to send or receive messages allowed by its active conversations. The plan mechanism is a generalization of the CSP guarded alternative construct. It allows processes to describe future as well as immediate intentions.

Conversation descriptions and plans contain application-specific information that can be used to enhance the efficiency of the application's communication. Other useful information can be derived from measurements made while the application is running. We present strategies for collecting and using information from these sources. These strategies attempt to use application-specific information to reduce the number of low-level messages needed to accomplish the application's communication.

We have implemented a version of the protocol generation system that supports application processes to be executed on the Crystal multicomputer. We describe several typical applications and evaluate their performance. We show that application-specific information from several sources can be used to significantly improve the efficiency of the application's communication.

Acknowledgements

I first of all thank my parents, Dale and Peg Rosenburg, who instilled in me my love of science and mathematics and who encouraged me to view graduate school, not as a hurdle to be overcome, but as an important part of my life. I thank my wife Julie for sharing this part of my life and for tolerating seven long years of student housing. I especially thank her for putting up with me during the couple months leading up to my final defense.

I thank the other members of the coffee-pot "syndicate", Aaron Gordon and Ennio Stacchetti, for many enjoyable afternoon discussions, some few of which may actually have contributed to my work here. I guess I get to keep the coffee pot, seeing as I'm the best procrastinator of the three of us. I especially thank my friend Bill Kalsow, with whom I've discussed every single minor aspect of my research (among other things). I can hardly claim any part of this research as completely my own. Bill's maintenance of the Modula-2 compiler and the TEX and LATEX typesetting software has also been invaluable.

Finally, I thank my advisor, Dr. Raphael Finkel, for shepherding me through this whole process and for periodically restoring my confidence in the work I was doing. I also thank Dr. Marvin Solomon for occasionally shaking that confidence and thereby improving the work.

This work was supported by the Charlotte project (DARPA contract numbers N00014-82-C-2087 and N00014-85-K-0788) and by a Tektronix Doctoral Fellowship.

Contents

List of Figures

Chapter 1

Introduction

The network hardware used to interconnect computers is inherently unreliable, and the interface it provides to programmers is not always convenient. Communication protocols that use checksums, timers, and acknowledgements to provide the user with an abstract reliable communication line are intended to mitigate these problems. However, communication protocols are notoriously difficult to design and debug, so most users of a distributed system rely on protocols provided by their installation's operating system. The available protocols may not conveniently and efficiently meet the needs of a particular application, but the programmer must often use a standard protocol because it is difficult to implement an application-specific protocol. Many researchers are trying to automate various phases of the task of implementing and validating new protocols.

This dissertation describes the development of a programming environment that allows a programmer to describe the communication requirements of an application at a very high level. Tools provided by the environment automatically construct reliable protocols that provide the services required by the application.

The programming environment provides mechanisms for describing both the

static form of conversations between application processes and the dynamic participation of those processes in conversations. These mechanisms let the programmer provide the communication system with detailed knowledge of the high-level communication requirements of an application. Other application-specific information can be derived from measurements made while the application is running. Knowledge from all these sources can be used to improve the efficiency of the application's communication.

Chapter 2 of this document describes related work in protocol specification and verification, and in distributed operating systems and programming languages for distributed systems. The user interface to our programming environment is presented in chapter 3. The language used to describe the static form of conversations is specified, as is the mechanism for participating in conversations. Chapter 4 discusses several issues that must be addressed by any implementation of the programming environment. Several strategies for using application-specific information to improve the efficiency of constructed protocols are described in chapter 5. Chapter 6 describes our implementation of the protocol generator for the Crystal multicomputer. This chapter also characterizes the performance of the constructed protocols, both in situations where the various improvement strategies are successful and in situations where they are not. Chapter 7 outlines several directions in which this project might be extended, and chapter 8 presents our conclusions.

Chapter 2

Related Work

The research presented here is not based directly on any previous work. The field most closely related to this project is that of protocol specification and verification. Work in interprocess communication languages and in operating systems that support message-based interprocess communication have also been highly influential.

2.1 Protocol Specification and Verification

Work in protocol specification and verification is aimed at making it easier to develop new and/or special-purpose protocols. Our hope is to allow the application programmer to specify a protocol at a level that is appropriate for the application, and to let the protocol generator handle the low-level details such as sequence numbers, acknowledgements, and retransmissions.

A large amount of work has been done in the general area of protocol definition and verification [Sunshine81, Danthine80, Rand-Corporation80]. Much of this effort has been directed toward the creation of formalisms in which protocols may be defined. There are two major parts of a protocol definition, the service specification and the protocol specification. A **service specification** defines the services

provided by a protocol to its users and specifies constraints on the use of those services. It does not define the method by which those services are to be provided. Some work has been directed toward formalizing service specifications. Bochmann [Bochmann80] describes a method for formally specifying local properties of a protocol interface and global protocol properties such as the requirement that messages be delivered reliably and in the order they were sent. Local properties are specified by providing a finite automaton that accepts legal sequences of client requests, and global properties are specified by providing a context-free grammar that constrains the joint behavior of two protocol clients. A **protocol specification** defines the entities that cooperate to provide the protocol services and specifies the interactions of those entities with each other and with their environment. Formal protocol specifications have been based on several models. These include finite automata [Bochmann80, Zafiropulo78, West78], Petri nets [Merlin76, Molloy82], and formal languages [Brand78, Good77, Nash83].

The goal of protocol verification is to show that a protocol specification correctly implements its corresponding service specification. Since formal service specifications are rarely available, many systems attempt to verify only general protocol properties such as reliability, freedom from deadlock and livelock, and completeness. A protocol entity is **complete** if it defines actions for all the input events that can occur in each of its states. In general, verifiers either exhaustively search the joint state space of the protocol entities [Bochmann80, West78], or they attempt to prove assertions about the behavior of the protocol [Thompson81, Good77, Brand78]. Exhaustive searches can discover problems such as the possibility of deadlock or the occurrence of events for which a protocol entity is not prepared, while assertion provers can guarantee protocol properties such as reliability. Recent work by Clarke, Emerson, and Sistla [Clarke86] combines these

approaches. Their system verifies temporal logic assertions about the behavior of communicating finite-state entities by exhaustively searching the joint state space of the entities. Related to protocol verification are the automatic testing of protocols [Linn83], and the analytic evaluation of protocol performance [Molloy82].

More closely related to our work are projects that attempt to automate various aspects of protocol specification. Chow, Gouda, and Lam [Chow85] describe a technique for constructing complex protocols from simpler components in such a way that the resulting complex protocols retain desirable properties of the components such as freedom from deadlock and completeness. The Automated Protocol Synthesizer of Ramamoorthy, Dong, and Usuda [Ramamoorthy85] allows a protocol designer to specify just one of the two communicating entities that comprise a protocol. Their synthesizer accepts a Petri-net description of a communicating entity and automatically creates a description of a corresponding entity. The resulting system is guaranteed to have desirable properties such as completeness, boundedness, and freedom from deadlock or livelock, provided the original entity satisfies certain easily-verified local constraints.

The projects most closely related to the work described here are those that formalize protocol specifications in such a way that defined protocols can be implemented automatically. Merlin [Merlin76] proposes the use of a Petri net interpreter to drive implementations of protocols described by Petri nets. Teng and Liu [Teng78] define a system that allows a designer to use context-free grammars to describe protocol entities, and that uses an automatic parser generator to construct implementations of defined protocols.

More recent (and more relevant) projects of this sort include a system developed by Sidhu and Blumer [Sidhu83] and a system developed by Anderson [Anderson85]. The protocol development system of Sidhu and Blumer allows a protocol designer to

use finite-state automata to describe protocol entities. For example, to describe the alternating-bit protocol [Tanenbaum81], the designer would define two automata, one to describe the sender and one to describe the receiver. Unfortunately, true finite automata need an unmanageably large number of states to represent many realistic protocols, and for that reason Sidhu and Blumer use finite automata extended with variables, predicates, and code fragments. Under this extension, a protocol entity is described by an automaton that has an explicit state, but that also has a collection of local variables it can use to hold additional state information such as sequence numbers, destination addresses, or message contents. Each transition in the automaton has associated with it an input event, a predicate on the local variables of the automaton, and a code fragment. Input events include such things as user requests, incoming messages, and timeouts. A transition is made only if its input event is available and its predicate evaluates to "true". The code fragment associated with a transition is executed when the transition is made. The code may change the values of local variables and generate output events.

Sidhu and Blumer provide a translator that consumes an extended automaton description and produces a set of tables that define the automaton. These tables are used by a protocol analyzer to verify standard protocol properties such as boundedness, completeness, and freedom from deadlock or livelock. They are also used to drive an implementation of the protocol. The automatically derived implementation depends on "interface-event" routines that must be provided by the protocol designer. These routines are used to encapsulate system-dependent details of the interfaces to the underlying communication system, to the protocol users, and to the operating environment. Sidhu and Blumer claim the coding of these routines is relatively straightforward and that the automatic construction of implementations is of great value in the design and testing of a new protocol.

Anderson's system allows a protocol designer to use attribute grammars to describe the behavior of protocol entities. The terminal symbols of such a grammar are names of interface events important to the protocol entity, and the grammar describes the set of all legal sequences of such events. Interface events include user requests, responses to user requests, incoming and outgoing messages, and timeouts. Each symbol in the grammar has an associated set of "attributes", and each production has an associated "enabling condition" and a set of "attribute assignments". Attributes are used to hold such things as sequence numbers, network addresses, and arguments to interface events. A top-down parsing technique is used, but a production is predicted only if its enabling condition is "true". When a production is predicted, its attribute assignments are evaluated. These assignments may change the attributes of any node in the parse tree, and may therefore have the effect of enabling other productions. Enabling conditions are used by a protocol entity to provide flow control, for example. Anderson uses a top-down parser to drive a protocol implementation. Conceptually, the parser maintains an incomplete parse tree that derives all the interface events that have already occurred. If a new user request becomes available or if an input event occurs, the parser attempts to extend the parse tree in such a way as to accept the new event. This extension may involve the expansion of productions that predict output events. In that case, a procedure that accomplishes the output event is called. As in Sidhu and Blumer's system, system-dependent details of the various interfaces are encapsulated in procedures provided by the protocol designer.

Anderson's approach has several advantages over that of Sidhu and Blumer. First, it provides a more powerful formalism for describing protocols. It allows the description of protocol features such as nested message sequences that can only be encoded artificially in the transition code fragments of the Sidhu and Blumer

model. Second, it allows the direct specification of the legal sequences of events, rather than the description of a machine that accepts all legal sequences. Protocol service specifications can be derived more easily from sequence descriptions than from automaton descriptions. Finally, an attribute grammar formally specifies the order of both input and output events. An automaton description includes a formal description of input events, but output events must be embedded in the code fragments attached to automaton transitions.

Both of these projects provide a framework in which a programmer can design a new protocol that will satisfy some desired service specification. Our hope is to generate protocols automatically from end-user service specifications. For our purposes, the service specification will be a description of the high-level conversation between application processes. As a trivial example, both of the systems described above can be used to implement the alternating-bit protocol. Our system might produce the alternating-bit protocol as output, given a conversation description that specifies the transmission of a unidirectional stream of messages from one process to another.

The reason so much work is being done in the area of protocol specification and verification is that communication protocols can be very complex programs. For this reason, it is too much to hope that the protocol generator proposed here will generate radically new, but correct, protocols. Instead it has the more modest goal of producing customized variants of existing protocols that perform well in given situations or protocols that dynamically adapt themselves to new conditions.

2.2 Interprocess Communication

This project is primarily concerned with the efficient implementation of message-based interprocess communication. To test the validity of some of our ideas, we have designed a communication mechanism that lets programmers explicitly describe the communication requirements of application processes. The design of this mechanism has been strongly influenced by the programming language CSP [Hoare78]. Our experience with the Charlotte distributed operating system [Finkel83] has also influenced the design.

This project achieves some measure of efficiency by avoiding the layered structure that characterizes many high-level protocols. Other operating systems and programming languages strive for efficiency in the same manner. The hierarchy of remote operations categorized and partially implemented by Spector [Spector82] is one example. The high-level language communication mechanism implemented by Leblanc [Leblanc82] is another. A third example is the remote procedure call facility provided by the V Kernel operating system [Cheriton83]. Saltzer [Saltzer84] argues for providing reliability only at the very highest level of layered systems. We accept his arguments to some extent, but we still provide reliable communication to the users of our system.

Chapter 3

Protocol System Specification

In this chapter we provide the specifications for a new message-based interprocess communication mechanism. This mechanism allows programmers to describe the communication behavior of application programs more explicitly than they can using existing interprocess communication mechanisms. We hope this mechanism is usable, but its elegance as a programming language has not been our primary concern. Instead we intend to explore ways to exploit the additional information the mechanism makes available to the communication system.

To use the protocol system, the programmer must first describe the conversations in which various processes will engage, and then program the active or passive participation of those processes in conversations.

3.1 Conversation Description

A **conversation description** is a static description of the form of a conversation between two application processes. It assigns names to the participants in the conversation, specifies the types of messages that the conversation requires, and includes a regular expression that describes all legal sequences of messages. Such

a description conforms to the syntax rules of Figure 3.1.

⟨*conv desc*⟩ ::= ⟨*conv name*⟩ ⟨*participants*⟩ : ⟨*reg expr*⟩

⟨*participants*⟩ ::= (⟨*participant*⟩ , ⟨*participant*⟩)

⟨*reg expr*⟩ ::= ⟨*reg expr*⟩ ⟨*reg expr*⟩
::= ⟨*reg expr*⟩ | ⟨*reg expr*⟩
::= ⟨*reg expr*⟩ *
::= { ⟨*reg expr*⟩ }
::= ⟨*primitive*⟩

⟨*primitive*⟩ ::= ⟨*msg class*⟩ ⟨*content type*⟩ : ⟨*direction*⟩

⟨*content type*⟩ ::=
::= (⟨*type list*⟩)

⟨*type list*⟩ ::= ⟨*type*⟩
::= ⟨*type*⟩ , ⟨*type list*⟩

⟨*direction*⟩ ::= ⟨*participant*⟩ --> ⟨*participant*⟩
::= ⟨*participant*⟩ <-- ⟨*participant*⟩
::= ⟨*participant*⟩ <-> ⟨*participant*⟩

⟨*conv name*⟩ ::= **id**
⟨*participant*⟩ ::= **id**
⟨*msg class*⟩ ::= **id**
⟨*type*⟩ ::= **id**

Figure 3.1: Conversation description syntax

The description heading begins with a name by which users can refer to the conversation description followed by the names of the two conversation participants in parentheses. These names may be any legal identifiers in the underlying language. (The degree to which these names may be overloaded will depend on the underlying language.) For example, a conversation between a fileserver and a client may begin with the heading:

```
FileProtocol (Fileserver, Client) :
```

The names `Fileserver` and `Client` may then be used within the body of the conversation description to refer to the conversation participants.

The heading is followed by a regular expression. The primitive elements of this expression describe individual messages. Each primitive begins with an identifier that specifies the class of this message. Again, a class name may be any legal identifier. Reasonable classes for the `FileProtocol` conversation might be `Open`, `Close`, or `Data`.

Following the message class is an optional parenthesized list of the data types that describe the contents of a message of this class. These data types can be any named type legal in the underlying language. If no data types are specified, then messages of this class have no content, and are used purely as signals. Messages of class `Open` might have contents of type `(FileName, FileMode)`, where `FileName` and `FileMode` are types declared elsewhere. A `Data` message might have contents of type `(DataArray)`, while a `Close` message might have no contents at all. A given message class may appear several times in a conversation description, but the message content types associated with all the appearances must be identical.

Finally, each primitive specifies the two participants between which this message is to be transmitted and the direction of the transmission. The participants are named, separated by one of the symbols "`-->`", "`<--`", or "`<->`". The symbols "`-->`" and "`<--`" show the direction the message is to travel, while the symbol "`<->`" specifies that a pair of messages of the given class is to be exchanged by the participants. A primitive that specifies the transmission of an `Open` message from the client to the fileserver might therefore be:

```
Open (FileName, FileMode) : Client --> Fileserver
```

Primitives may be combined using the regular expression operators concatenation, alternation, and arbitrary repetition. Concatenation is specified by lexically

concatenating the operands, alternation by an infix "|", and arbitrary repetition by a postfix "*". Repetition has higher precedence than concatenation, which in turn has higher precedence than alternation. These precedences are not reflected in the ambiguous grammar of Figure 3.1. Operands of these operators may be delimited with "{" and "}" to override the default precedences. A simple conversation between a client and fileserver might have the following description:

```
FileProtocol (Fileserver, Client) :
    {
        Open (FileName, FileMode) : Client --> Fileserver
        {
            Data (DataArray) : Client <-- Fileserver
        } *
        Close : Client --> Fileserver
    }
```

This conversation description requires the client to send an `Open` message to the fileserver, which will then send an arbitrary number `Data` messages back to the client. The client must then send a `Close` message to the fileserver. A more complex description that allows data to be both read and written and that allows seeking in the file might be:

```
FileProtocol (Fileserver, Client) :
    {
        Open (FileName, FileMode) : Client --> Fileserver
        {
            Data (DataArray) : Client <-- Fileserver
            |
            Data (DataArray) : Client --> Fileserver
            |
            Seek (integer) : Client --> Fileserver
        } *
        Close : Client --> Fileserver
    }
```

As a second example, a conversation between a producer and a consumer might consist of an arbitrarily long sequence of messages, each of class `Item` and having

contents of type (ItemType). Such a conversation would have the following description:

```
StreamProtocol (Producer, Consumer) :
    {
        Item (ItemType) : Producer --> Consumer
    } *
```

A remote procedure call style of process communication might be specified by the following conversation description:

```
RpcProtocol (Client, Server) :
    {
        Question (QuestionType) : Client --> Server
        Answer   (AnswerType)   : Client <-- Server
    } *
```

3.2 Conversation Participation

A conversation description is a static description of the form a conversation will take. It is completely specified at the time a protocol for it is generated. The uses to which processes want to put conversations, on the other hand, are not known until the processes are running. A mechanism that allows processes to participate in conversations must therefore be provided. The protocol generator takes a conversation description and builds procedures that processes may use to participate in the conversation. Some of these procedures are specific to a particular conversation, while others are applicable to all conversations.

The function NewConversation is provided for the purpose of initiating new conversations. The first argument to this function is a conversation name as it appears in the header of some conversation description, the second argument is one of the participant names of the header, and the remaining argument is the name of the process that will be the other participant in the conversation. The

function returns a **conversation descriptor** that may be used later to refer to the conversation. For example, if a process wants to take the part of the client in a `FileProtocol` conversation, it must make the call:

```
conv := NewConversation (FileProtocol, Client, flsrvr);
```

This call initiates a new `FileProtocol` conversation in which the calling process is to be the client and the fileserver is to be the process designated by `flsrvr`. The nature of this process designation will depend on the underlying communication facility. It may be as simple as a machine address if the system supports only one process per machine. The other participant in a new conversation must make a matching initialization call. Two calls to `NewConversation` match if they specify the same conversation description but different participant names and if the initiating processes name each other as participants. A process may participate in several conversations simultaneously. `NewConversation` is a non-blocking function. The conversation descriptor it returns may be used immediately, but no messages will be transferred until the new conversation is actually established. If two processes open several identical conversations between themselves, their `NewConversation` calls will be paired in the order they are made. Conversations are closed with the procedure `CloseConversation`, which takes a conversation descriptor as its only argument.

Once conversations have been initialized, processes will from time to time want to send or receive messages allowed by the conversation descriptions. Processes communicate by building and executing "plans". A **plan** is a regular expression whose primitive elements are "operations". An **operation** is either a "communication event" or an "action" procedure call. A **communication event** (or **event**) describes the transmission or the reception of a message or the exchange of a pair of messages. It specifies the message's conversation and class and describes the

process data areas that are to be read and/or written. **Actions** are application procedures that are to be called at specific points during the execution of plans. Since a plan is a regular expression, it may specify a sequence of operations, a set of alternative operations, an operation to be executed repeatedly, or some combination of these structures. For example, a client in a `FileProtocol` conversation may build a plan consisting of a single communication event, the reception of a `Data` message from the fileserver. The fileserver, on the other hand, must be willing to perform one of several different operations, depending on the wishes of the client. It must simultaneously be willing to send a `Data` message, receive a `Data` message, receive a `Seek` message, or receive a `Close` message. It must therefore build a plan that includes these four events as alternatives. The particular alternative that is chosen will depend on the wishes of the client at the other end of the conversation. A plan may include events that belong to different conversations, and a process may build and use several plans simultaneously. However, no single conversation may be used with more than one plan. Plans execute independently, and allowing different plans to involve the same conversation would destroy this independence.

At runtime, application processes use **plan-constructor** procedures to build plans. The function `NewPlan` is used to create a new plan. It takes a single argument, an arbitrary integer called the **plan identifier**. This argument is passed to action procedures that are called during execution of the plan. An action procedure can use it to identify the plan that caused the procedure to be called. `NewPlan` returns a **plan descriptor** that may be used to refer to the plan.

A newly created plan is empty, that is, it does not specify any operations to be performed. Regular expressions that specify operations may be appended to a plan, and the completion of expressions may be awaited. The routines `Append` and `Wait` that accomplish these functions are defined later. Plans may be deallocated

using the procedure `ClosePlan`, which takes a plan descriptor as its only argument. A plan may be closed only when all appended expressions have been awaited.

The primitive elements of these regular expressions are created by the function `Action` and by functions of the form `Put`⟨*class*⟩, `Get`⟨*class*⟩, and `Trade`⟨*class*⟩, where ⟨*class*⟩ is some message class that appears in a conversation description. These functions return a pointer type that we will call a **regular expression.** `Action` takes as its only argument a procedure name, and the regular expression it returns describes a call to the named procedure. The first argument of the `Get`⟨*class*⟩, `Put`⟨*class*⟩, and `Trade`⟨*class*⟩ functions is a conversation descriptor returned by a `NewConversation` call. The remaining arguments correspond to the data types specified for the contents of messages of class ⟨*class*⟩, one argument for each specified type.

These primitive actions are combined using the functions `Sequence`, `Choice`, and `Cycle`, which take regular expressions as arguments and return new regular expressions. For notational convenience, we allow `Sequence` and `Choice` to take an arbitrary number of arguments. A particular regular expression may be incorporated in a larger expression only once, so that expressions may be internally represented as trees. A procedure `FreeExpression` can be called to deallocate an arbitrarily complex regular expression.

To continue our `FileProtocol` example, a client's regular expression to simply open a file might be built by the single function call:

```
expr := PutOpen (conv, "myfile", filemode);
```

A regular expression to open a file, read one block, and close the file could be constructed with the following expression:

```
expr := Sequence (
            PutOpen (conv, "myfile", filemode),
            GetData (conv, buffer),
            PutClose (conv));
```

The fileserver might use the following regular expression to handle an open request, an arbitrarily long sequence of read, write, and seek requests, and a close request:

```
expr := Sequence (
            GetOpen (conv, filename, filemode),
            Action (Open),
            Cycle (
                Choice (
                    Sequence (
                        PutData (conv, buffer),
                        Action (Read)),
                    Sequence (
                        GetData (conv, buffer),
                        Action (Write)),
                    Sequence (
                        GetSeek (conv, addr),
                        Action (Seek)))),
            GetClose (conv),
            Action (Close));
```

Here `Open`, `Read`, `Write`, `Seek`, and `Close` are procedures provided by the fileserver that perform the indicated file operations. Since they are passed no arguments other than a plan identifier, they must find such information as file names, buffers, and seek addresses in variables global to themselves.

Once a regular expression has been constructed, it can be added to a plan using the procedure `Append`. `Append` takes two arguments, a regular expression and a plan descriptor, and it adds the regular expression to the end of the plan. The expression will then be executed as soon as any previous regular expressions in the plan have been satisfied. Application processes use the `Wait` function call to block themselves while plans are executing. `Wait` takes no arguments. Each call to `Wait` returns a single regular expression that was previously appended to a plan. All

the regular expressions appended to a particular plan will be returned in the order in which they were appended, but expressions appended to different plans may be arbitrarily interleaved. An expression returned by `Wait` can be re-executed by appending it once again to a plan, or it can be deallocated with `FreeExpression`.

The procedure `Sleep` blocks the application process until all outstanding regular expressions have been completed. It calls `Wait` repeatedly and deallocates the regular expressions that are returned.

Finally, the procedure `Perform` is provided as an abbreviation for a common sequence of program actions. It takes a regular expression as its only argument. It appends the regular expression to an anonymous plan, waits for the expression's completion, and then deallocates the expression. This procedure allows most applications to avoid dealing directly with plans. `Perform` may be called only when there are no outstanding regular expressions.

Action procedures are not executed concurrently, either with each other or with the main process. They will be executed only while the main process is blocked in a call to `Wait`, `Sleep`, or `Perform`. Action procedures may not make any of these blocking calls, but they may build regular expressions and append them to plans.

We can now provide an operational definition of the semantics of plan execution. Execution of a regular expression can best be described in terms of the expression's equivalent deterministic finite automaton (DFA). Well known algorithms [Aho77] may be used to convert a regular expression into an equivalent DFA whose transitions are labeled with the action procedures and communication events of the expression. For the purposes of this conversion, all the primitive elements of the expression are considered to be distinct. DFA states that have only one exit transition are called **decisive** states, and the action procedure or communication event labeling the single transition is said to be **decisively specified.** A regular expres-

sion is executed by following a path through its DFA and accomplishing the action procedures and communication events that label the path.

In this discussion, we will assume each plan is executed by an abstract entity called the plan's **driver**. Drivers sequentially execute the regular expressions that are appended to their plans. Each driver maintains a DFA state variable and attempts to accomplish one of the operations that label the exit transitions of the current state. The current state is advanced after each operation is accomplished.

We impose two restrictions on regular expressions to simplify their execution. First, all action procedures must be specified decisively. We make this restriction so that drivers never have to choose between alternative action procedures or between an action procedure and a communication event. Second, the communication events that label the transitions leaving any given DFA state must be mutually distinguishable. Two communication events are indistinguishable if they specify the same conversation and the same message class and direction. Events may be indistinguishable and yet not identical because they specify different data areas to be read or written. Drivers have no basis for choosing among indistinguishable events. Regular expressions that violate either of these restrictions are said to be **ambiguous**, because they force drivers to make impossible choices. Figure 3.2 contains a few examples of ambiguous regular expressions.

We also require regular expressions to terminate properly. A regular expression **terminates properly** if no final state of its equivalent DFA has a nonempty set of exit transitions. This requirement allows drivers to easily detect the completion of regular expressions. Its practical effect is to disallow regular expressions that end with cycles.

Drivers may call action procedures whenever they are encountered during the execution of regular expressions, subject to the concurrency constraints mentioned

```
Choice (
    Action (ProcedureA),
    Action (ProcedureB))

Choice (
    PutOpen (conv, "FilenameA"),
    PutOpen (conv, "FilenameB"))

Sequence (
    Cycle (
        GetData (conv, buffer)),
    GetData (conv, buffer))

Sequence (
    Cycle (
        PutData (conv, buffer)),
    Action (Close))
```

Figure 3.2: Ambiguous regular expressions

earlier. The execution of an expression may be continued as soon as a called action procedure returns.

Drivers must cooperate to discover and accomplish matching pairs of communication events in their plans. The events in a matching pair may be accomplished only when both events label transitions leaving the current states of their respective DFA's. Two communication events match if they occur at opposite ends of a conversation, they specify the same message type, their transfer directions are consistent, and at least one of them is specified decisively. Two transfer directions are consistent if one specifies a send and the other specifies a receive, or if both specify an exchange.

The requirement that at least one of every matching pair of communication events be specified decisively allows an efficient implementation of the matching procedure. Drivers must commit themselves to the accomplishment of decisively

specified communication events that are encountered during the execution of their plans. A driver facing a choice among alternative events can simply await a commitment from one of its colleagues.

Figure 3.3 outlines the algorithm drivers follow in executing plans. This algorithm is intended to clarify the semantics of plan execution, not to serve as an implementation blueprint.

```
procedure PlanDriver (Plan).
    while (Plan has not been closed) do
        Await an appended expression from the client.
        Build the expression's equivalent DFA.
        Assign the DFA's start state to CurrentState.
        while (CurrentState is not a final state) do
            if (CurrentState is decisive) then
                if (the transition's label is an action procedure) then
                    critical section
                        Call the action procedure.
                    end
                else {the transition's label is a communication event}
                    Commit to the decisively specified event.
                    Await a corresponding commitment.
                    Complete the event.
                end
            else {CurrentState is not decisive}
                Await a commitment to one of the alternative events.
                Commit to the matched event.
                Complete the event.
            end
            Advance CurrentState.
        end
        Return the completed expression to the client.
    end
end PlanDriver.
```

Figure 3.3: A plan driver's algorithm

The complete fileserver-client example appears in Figures 3.4-3.6. The client

and server processes are written in a Modula-like language.

```
FileProtocol (Fileserver, Client) :
    {
        Open (FileName, FileMode) : Client --> Fileserver
        {
            Data (DataArray) : Client <-- Fileserver
            |
            Data (DataArray) : Client --> Fileserver
            |
            Seek (integer) : Client --> Fileserver
        } *
        Close : Client --> Fileserver
    }
```

Figure 3.4: FileProtocol example – conversation description

```
process client (flsrvr : ProcessDesignator);
var conv : Conversation;
    buffer : DataArray;
begin
    conv := NewConversation (FileProtocol, Client, flsrvr);
    Perform (PutOpen (conv, "myfile", READ+WRITE));
    Perform (GetData (conv, buffer));
    Perform (GetData (conv, buffer));
        .
        .
    Perform (PutSeek (conv, seekaddr));
    Perform (GetData (conv, buffer));
        .
        .
    Perform (PutSeek (conv, seekaddr));
    Perform (PutData (conv, buffer));
        .
        .
    Perform (PutClose (conv));
end client;
```

Figure 3.5: FileProtocol example – client process

```
process fileserver (clnt : ProcessDesignator);
var conv : ConversationID; name : FileName; mode : FileMode;
    fd, addr : integer; buffer : DataArray;

    procedure Open (pid : PlanIdentifier);
    begin fd := open (name, mode); read (fd, buffer) end Open;

    procedure Read (pid : PlanIdentifier);
    begin read (fd, buffer) end Read;

    procedure Write (pid : PlanIdentifier);
    begin write (fd, buffer); read (fd, buffer) end Write;

    procedure Seek (pid : PlanIdentifier);
    begin seek (fd, addr); read (fd, buffer) end Seek;

    procedure Close (pid : PlanIdentifier);
    begin close (fd) end Close;

begin {fileserver main process}
    conv := NewConversation (FileProtocol, Fileserver, clnt);
    Perform (
      Sequence (
        GetOpen (conv, name, mode),
        Action (Open),
        Cycle (
          Choice (
            Sequence (PutData (conv, buffer), Action (Read)),
            Sequence (GetData (conv, buffer), Action (Write)),
            Sequence (GetSeek (conv, addr),   Action (Seek)))),
        GetClose (conv),
        Action (Close)));
end fileserver;
```

Figure 3.6: FileProtocol example – server process

3.3 Design Rationale

The design presented here is the result of a continuing effort. It incorporates a few arbitrary decisions, but most of the choices have been made with some justification.

3.3.1 Conversations

An early design decision we had to make was the choice of an appropriate language for describing conversations. At one point we considered using context-free grammars, but we soon found that all the example conversations we tried to describe were regular languages. Conversation descriptions only constrain the sequence of message types, not the contents of messages, so non-regular concepts such as sequence numbers or message identifiers can easily be included in the contents. Since the power of context-free grammars appeared to be unnecessary, and since regular expressions are more concise than context-free grammars for describing regular languages, we elected to use regular expressions to describe conversations.

A second choice we made was to limit conversations to two participants. Allowing an arbitrary number of participants in a conversation would be desirable, but the semantics of conversations and plans would be considerably more complicated. The possibility of initiating conversations with only some of the participants present, and of allowing participants to come and go during the course of a conversation, would have to be considered. Also, the meaning of consecutive operations that involve disjoint subsets of the participants would have to be defined. Fortunately, many conversations naturally involve two participants, and interactions among larger groups can be accomplished with pair-wise conversations.

We chose to design a primitive mechanism for establishing new conversations. It requires the two participants of a new conversation to name each other explicitly.

It does not provide any sort of "passive" open such as is available in TCP [Postel81], nor does it provide a mechanism for establishing new connections based on existing connections as is available in the Charlotte operating system [Finkel83]. The only complicated aspect of the mechanism is that it is specified to be nonblocking. This requirement complicates the implementation somewhat, but it allows application processes to be much less careful of the order in which conversations are initialized than would be the case if `NewConversation` were a blocking function. For example, each process in a ring of processes can open a conversation first with its left neighbor and then with its right neighbor without worrying about deadlock. Establishing conversations is not an important aspect of this research project, so we designed as simple a mechanism as possible.

3.3.2 Plans

In designing a mechanism for participating in conversations, we limited ourselves to a purely procedural interface in hopes of avoiding both the religious issues involved in creating a new interprocess communication language and the effort of implementing a compiler for an entire programming language. Otherwise we would probably have adopted a syntax similar to that of CSP [Hoare78]. The plan mechanism we developed has all the power of the CSP communication constructs but is more flexible.

In CSP, processes involved in a message transmission must name each other explicitly. In our system, process names determined at runtime are used to open conversations, and individual messages refer to conversations rather than processes. A pair of processes may be connected by several conversations, allowing multiple channels between processes.

All communication is synchronous in CSP. When a process tries to send or

receive a message it is blocked until the reciprocating process attempts a corresponding operation. The same synchronous communication is available in our system using the `Perform` procedure, but the programmer also has the option of separating the notification of an operation's completion from the submission of the operation request. Multiple plans executing concurrently allow an additional degree of asynchrony.

The power of CSP's alternative construct with input and/or output guards is provided by our system's `Choice` plan constructor. Our system's `Sequence` and `Cycle` constructors provide additional expressive power. They allow a process to specify its intentions more completely, thereby providing more information that may be used to improve communication efficiency.

Unfortunately, algorithms can sometimes be expressed more elegantly in CSP than in our system. In CSP, a communication guard may be combined with a Boolean expression to enable or disable a particular clause of an alternative construct. We have no similar mechanism. Since plans are constructed at runtime, it is possible to include only the clauses that should be enabled, but this solution is not particularly elegant. In addition, an input or output guard in a CSP alternative construct is lexically adjacent to the code fragment that is to be executed if the communication operation is accomplished. In our system, the code fragment, no matter how trivial, must be embedded in an action procedure. The resulting proliferation of tiny procedures can lead to a loss of clarity.

We turn now to a discussion of the reasons for imposing the various restrictions on regular expressions that were described in the last section. These restrictions include the requirements that regular expressions be unambiguous and properly terminating and the requirement that at least one of the communication events in each matching pair be specified decisively.

Ambiguous regular expressions are disallowed because they force drivers to make decisions that cannot be made rationally without knowledge of the future course of a conversation. Suppose alternative paths in a regular expression's equivalent DFA were allowed to begin with action procedures. A driver would then have to look ahead past the action procedures to later communication events to choose a path consistent with the commitments of other drivers. The problem is more difficult if alternative paths are allowed to begin with indistinguishable communication events, because then drivers would have to look ahead past communication events as well as action procedures to choose correct paths. Ambiguous expressions could be handled correctly using the standard algorithm for constructing a DFA from a nondeterministic finite automaton (NFA), but the increase in power would not be worth the additional effort. If a regular expression is unambiguous, the construction of its equivalent DFA is straightforward because the usual intermediate NFA is not required.

Regular expressions are required to terminate properly so that the completion of an expression occurs at a well-defined time and so that the decisiveness of a given communication event can be readily determined. Suppose we allow regular expressions whose equivalent DFA's have final states with nonempty sets of exit transitions. A driver that reaches such a final state in a DFA cannot decide whether to continue executing the current expression or to move on to the next expression until it receives a commitment from another driver. An application process could conceivably await the completion of the current expression before initiating the chain of events that leads to the necessary commitment. Deadlock would be the result. Of course, application processes can become deadlocked without going to nearly so much trouble, so decreasing the possibility of deadlock is not a strong justification for the proper termination requirement. A better justification is that

the requirement eliminates the confusion that can arise when a final state of an expression's equivalent DFA has exactly one exit transition. Suppose an application process builds a plan consisting only of a cycle that specifies the arbitrary repetition of a single communication event. Is that single event specified decisively? The programmer may well think so since there are no other events that are possible alternatives, but in fact the plan's driver can never commit itself to accomplishing the event because another expression may be appended to the plan at any time. The proper termination requirement disallows this expression and all expressions that end with cycles.

We make the requirement that at least one of the communication events of each matching event pair be specified decisively so that plans can be executed efficiently. Without this restriction, plans would have all the power and all the implementation difficulty of CSP with both input and output guards. Several algorithms for implementing full CSP have been published [Bernstein80, Buckley83], and any of these algorithms can be used to implement plans. Unfortunately, they all require drivers to sequentially poll the other drivers involved in alternative communication events, so the number of messages required to accomplish a single event depends on the number of alternatives. Moreover, all the algorithms assume an underlying reliable communication service, making them even less efficient. An alternative approach is to restrict the use of plans to make them more efficiently implementable. CSP itself as defined by Hoare disallows output guards, allowing an implementation that requires just two reliable messages to accomplish each client transmission. Unfortunately, this restriction breaks the symmetry of input and output, making it difficult to express some algorithms naturally [Hoare78]. The restriction we have adopted maintains this symmetry but still allows an efficient implementation.

We do not expect this project to stand or fall on the basis of the design of plans.

The major drawback of plans is that programs that use them are not as clear as programs that use CSP alternative constructs. However, plans are more powerful than alternative constructs, and they allow a process to provide more detailed knowledge of its intentions to the system. They allow regular expression operators other than alternation, and they are constructed at runtime, so the number and contents of clauses can be varied as a program executes. It should be fairly easy to write a preprocessor that will turn a CSP program into a program that uses conversation descriptions and plans. A preprocessor might also be used to provide a more pleasant programming language syntax for dealing with plans.

Chapter 4

Abstract Implementation

The programming system described in chapter 3 presents several interesting implementation problems, even without considering the use of application-specific information to improve the efficiency of the application's communication. In the following discussion, we will assume the top-level application is embodied in a collection of **client** processes. Each client is assigned an **agent** whose purpose is to communicate with other agents on the client's behalf. Agents establish conversations between clients and interpret client plans. The internal structure of agents will be left unspecified, but we will assume agents reside in an environment that provides a rudimentary communication service that will allow one agent to send a message to another with some degree of success, and a timer mechanism by which an agent can be notified when a certain amount of time has elapsed. Agents can use these services to implement reliable communication channels among themselves, and the reliable channels can in turn be used to implement client services. However, this layered approach can be inefficient, so client services for which efficiency is critical should be implemented directly with the low-level facilities.

4.1 Establishing Conversations

Agents may use layered reliable communication channels to establish conversations, assuming new conversations are established relatively infrequently so that efficiency is not critical. Initializing each new conversation requires the exchange of a pair of reliable messages describing the clients' corresponding `NewConversation` function calls.

Each agent must maintain two lists of half-open conversations, one of locally open conversations and another of remotely open conversations. When an agent's client calls `NewConversation`, the agent creates a new locally open conversation to preserve the function's arguments and returns its descriptor to the client. It then sends an initialization message describing the arguments to the agent of the process designated to be the conversation's other participant. An agent that receives such a message creates a new remotely open conversation. Agents attempt to pair locally open conversations with remotely open conversations according to the matching criteria specified in section 3.2. Some care must be taken to ensure that multiple conversations between a single pair of clients are paired consistently. When two matching half-open conversations are discovered, they can be merged to form a fully open conversation.

Since `NewConversation` is nonblocking, clients may build regular expressions involving conversations that are still only locally open. An agent that encounters such a conversation during the execution of a plan must suspend the execution until the conversation becomes fully open.

4.2 Interpreting Plans

In addition to establishing conversations, agents serve as the plan drivers for all of their clients' active plans. Drivers follow the algorithm of Figure 3.3, but the mechanics of making and awaiting commitments must be discussed in more detail. We associate with each end of all open conversations a new abstract entity called a **conversation server**. Plan drivers use the servers associated with their open conversations to accomplish communication events. The servers at opposite ends of a conversation communicate with each other to accomplish the data transfers specified by pairs of matching communication events.

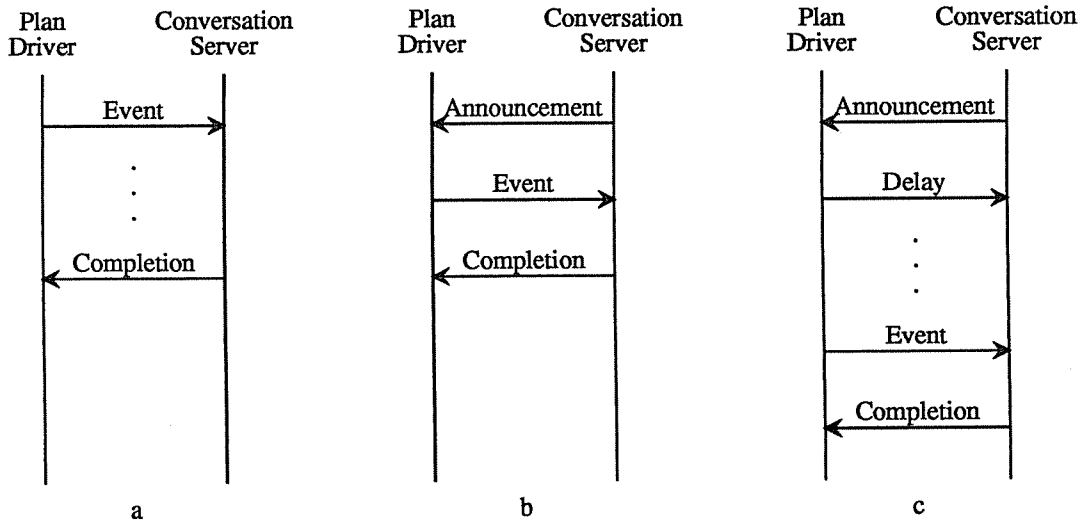

Figure 4.1: The plan driver–conversation server interface

Figure 4.1 illustrates the interface between a plan driver and each of its associated conversation servers. When the driver encounters a decisively specified communication event, it gives the event to the appropriate server and awaits the event's completion (Figure 4.1a). When a server learns that its partner has been committed to the accomplishment of a particular event, it announces that fact to its plan driver. The announcement may allow the driver to choose one of sev-

eral alternative communication events, in which case the driver gives the matched event to the server and awaits the event's completion (Figure 4.1b). Otherwise the driver remembers the announcement but tells the server the remote event cannot be matched immediately (Figure 4.1c). When a plan driver encounters a branch point in its plan, it checks its servers' outstanding announcements to see if any of the alternative events can be accomplished immediately. If so, the driver gives the matched event to the appropriate server and awaits the event's completion (Figure 4.1c). Otherwise the driver waits for new announcements from its servers (Figure 4.1b). (If several alternative events match outstanding announcements, the driver chooses one of them arbitrarily.)

The servers at opposite ends of a conversation must cooperate to provide the services described in the last paragraph. In effect, they provide a communication channel that has a rather peculiar service specification. We define two protocols that a pair of conversation servers may use to meet this specification. The EXCHANGE protocol depends on an underlying reliable communication service, while the TRIPLE protocol depends only on low-level communication facilities.

4.2.1 The EXCHANGE Protocol

Two conversation servers following the EXCHANGE protocol exchange reliable messages to accomplish each matching pair of communication events. An **event** message describing one of the events and including any specified data is transmitted in each direction. The exchanged messages may occur in either order, and they may cross on the communication medium.

When an otherwise idle server gets a decisively specified communication event from its plan driver, it gathers any outgoing client data specified by the event and sends its partner an **event** message describing the event. When it receives a

corresponding **event** message it scatters any incoming client data and notifies the driver of the event's completion.

When an otherwise idle server receives an **event** message, it announces the decisively specified remote event to its plan driver. If the driver responds with a matching event, the server accomplishes the event and sends an **event** message to its partner. Otherwise the server waits. When it later gets a matching event from its driver, it accomplishes the event and sends an **event** message to its partner. In either case the server must notify the driver of the event's completion as soon as it has sent the **event** message.

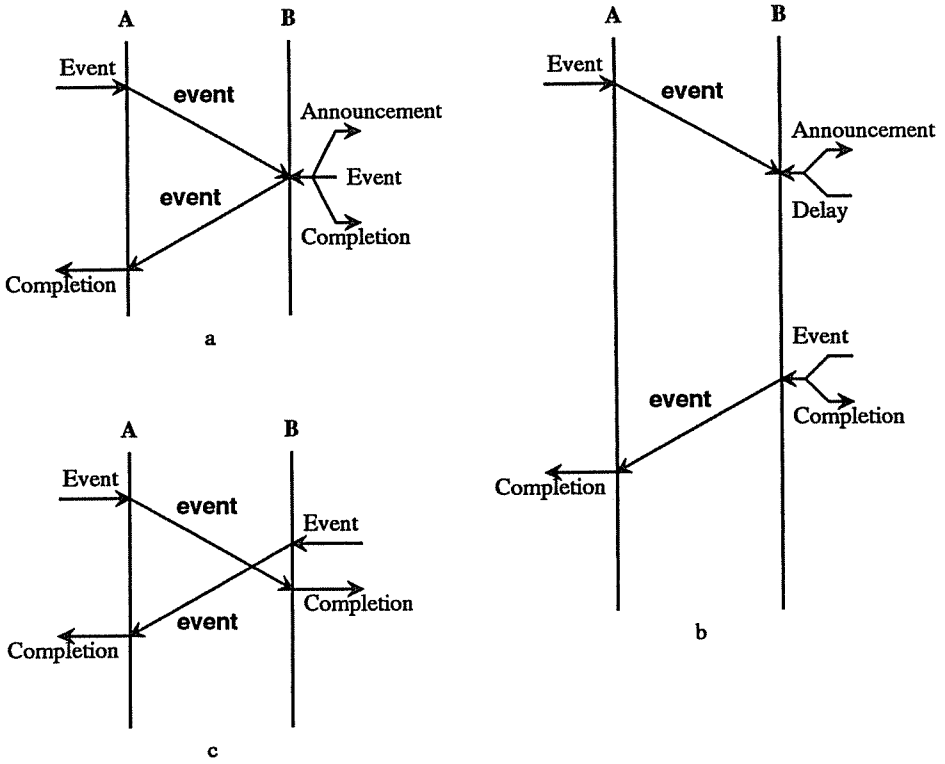

Figure 4.2: Transactions of the EXCHANGE protocol

The sequence of messages a pair of servers uses to accomplish a particular communication event is called a **transaction.** Figure 4.2 illustrates the three types of transaction that can occur in the EXCHANGE protocol. In this and later figures,

pairs of parallel vertical lines represent time in two cooperating communication servers. Labeled arrows between the time lines indicate transmitted messages, while arrows outside the lines indicate interactions between the servers and their plan drivers.

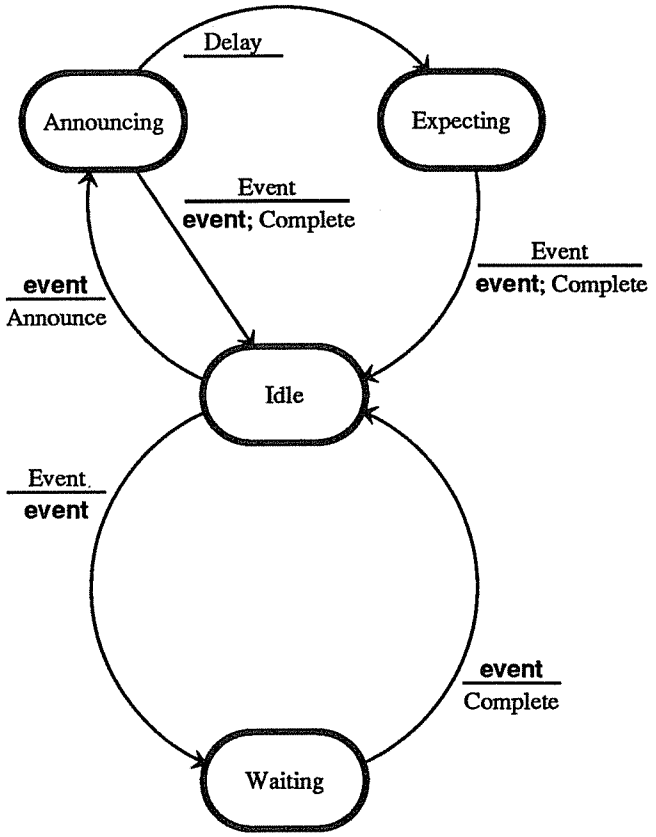

Figure 4.3: The EXCHANGE protocol DFA

A conversation server following the EXCHANGE protocol can be specified as the deterministic finite automaton shown in Figure 4.3. In this diagram labeled ovals represent the states of the DFA and labeled arcs represent the DFA transitions. Each transition label consists of two parts separated by a horizontal line. Above the line is the DFA input that causes the transition. Possible inputs are **event** messages from the server's partner and "Event" or "Delay" notifications from the server's plan driver. Below the line is the action to be taken by the automa-

ton when the transition is taken. Possible actions include sending messages and generating "Announcement" or "Completion" notifications. Sending a message is specified simply by naming the message type, while the actions "Announce" and "Complete" generate appropriate notifications for the server's plan driver.

4.2.2 The TRIPLE Protocol

The major drawback of the EXCHANGE protocol is that it depends on an underlying reliable communication service, at some cost in efficiency. To overcome this drawback we present the TRIPLE protocol, which depends only on unreliable low-level messages and on timers. It covers the function of both the EXCHANGE protocol and the underlying reliable message protocol.

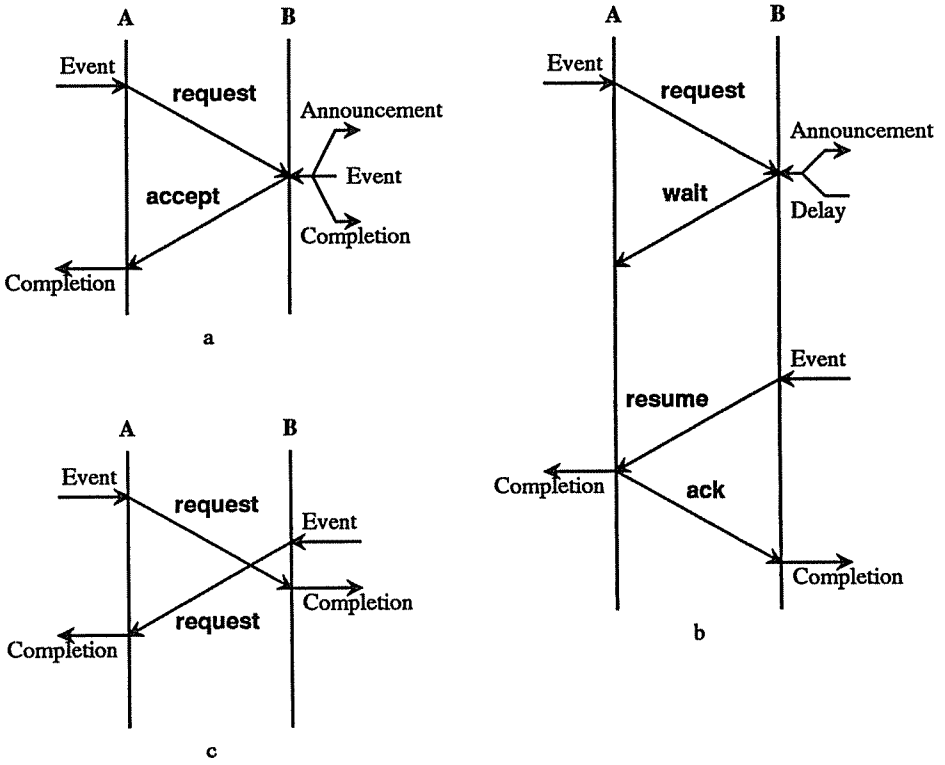

Figure 4.4: Normal transactions of the TRIPLE protocol

Figure 4.4 illustrates the TRIPLE protocol's three principle transaction types.

Suppose **A** and **B** are the servers at opposite ends of a conversation, and that **A** gets a decisively specified communication event from its plan driver. Then **A** sends **B** a **request** message describing the event. Upon receiving the **request**, **B** announces the event to its plan driver. If the driver responds with a matching event, **B** returns an **accept** message describing the matching event to **A** (Figure 4.4a). Otherwise **B** responds with a **wait** message. At some later time **B**'s plan driver provides a matching event. Then **B** sends a **resume** message to **A**, and **A** responds with an **ack** message (Figure 4.4b). **A** notifies its driver of the original event's completion either when it receives an **accept** message or when it receives a **resume** message. **B** notifies its driver of the matching event's completion as soon as it has sent the **accept** or **resume** message.

The **request** message carries data to be transferred from **A**'s client to **B**'s client for this event. The **accept** message or the **resume** message carries data to be transferred the other direction.

Since both communication events of a matching pair may be specified decisively, matching **request** messages may cross on the transmission medium. In that case, each server treats the **request** message it receives as if it were an **accept** message, and each proceeds to the next transaction (Figure 4.4c).

Figure 4.5 illustrates two transactions that involve lost messages. Retransmission timers and sequence numbers are used to recover from such failures. After sending a **request** message, a server starts a retransmission timer. If the timer expires before the sender has received either an **accept** or a **wait** message, then the **request** message is sent again. Essentially, the **request** is a reliable message, and the **accept** or **wait** is an acknowledgement that it was received (Figure 4.5a,b). Similarly, a server starts a retransmission timer after sending a **resume** message, and retransmits the **resume** if the timer expires before an **ack** is received. The length

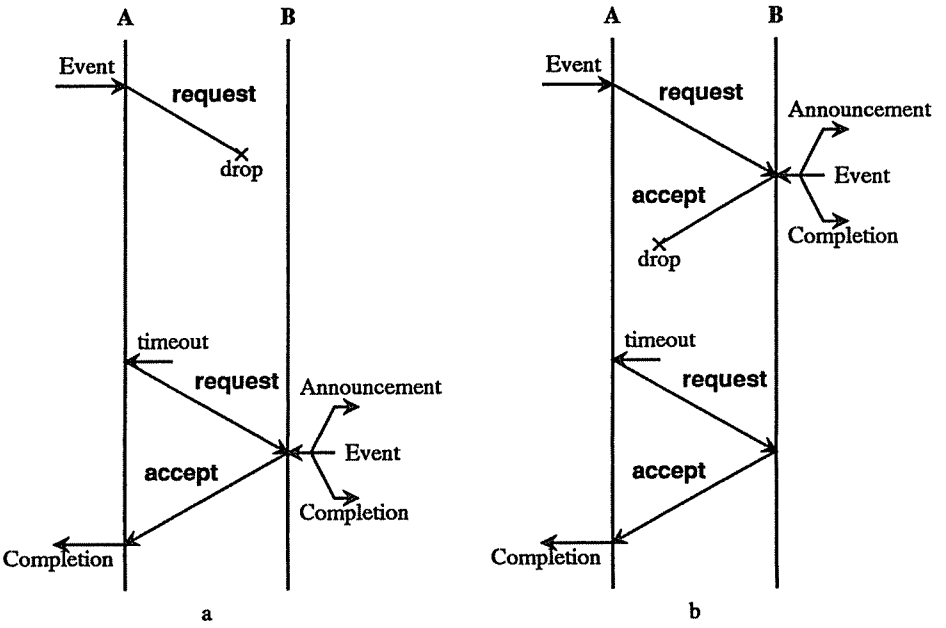

Figure 4.5: Some abnormal TRIPLE protocol transactions

of time a server will wait before retransmitting a **request** message is a protocol parameter. Another parameter controls the retransmission of **resume** messages.

Sequence numbers are used to detect duplicate messages. A server that receives a duplicate **request** message responds with another **wait** message if the matching event is not yet available. Otherwise it resends its previous **accept** message (Figure 4.5b). Since **accept** messages may carry data, agents must always remember the last data message they sent. A server that receives a duplicate **resume** message responds with another **ack** message. Duplicate **wait**, **accept**, or **ack** messages are ignored.

A consequence of the fact that **request** messages may cross is that two-valued or "alternating-bit" [Tanenbaum81] sequence numbers are not adequate. A server that has sent a **request** message may receive a **request** message from the current transaction (Figure 4.6a), from the previous transaction (Figure 4.6b), or from the next transaction (Figure 4.6c), and it must respond appropriately in each case. All

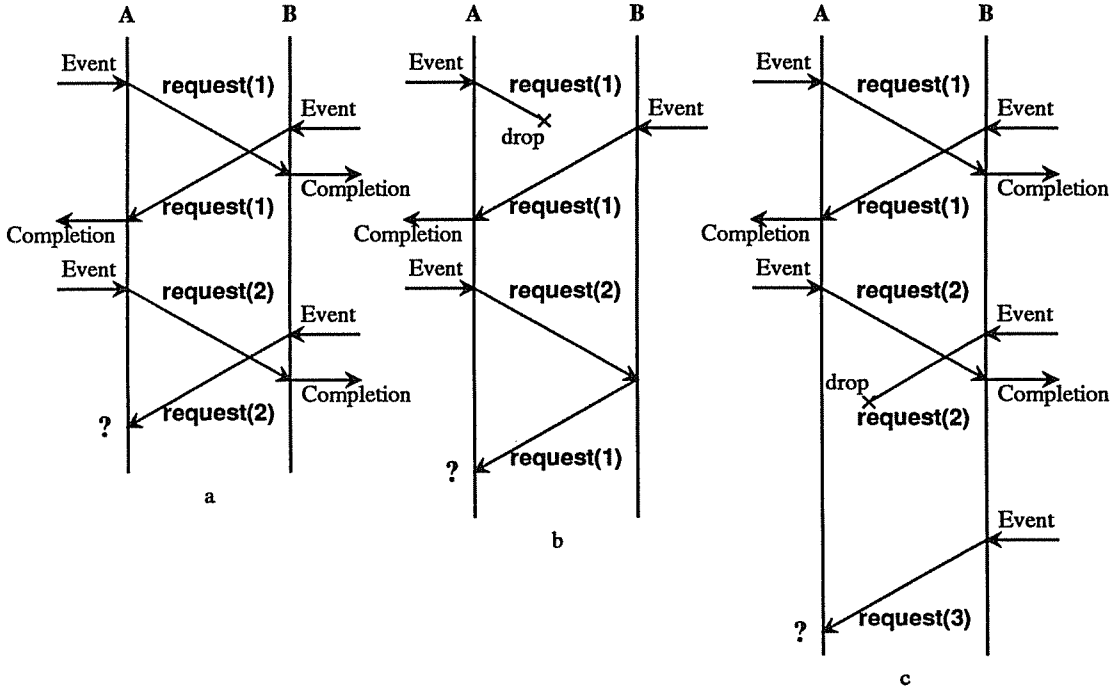

Figure 4.6: The necessity of three-valued sequence numbers

three scenarios in Figure 4.6 appear identical to server **A**, except that the sequence numbers on the last message **A** receives in each scenario are different. Three-valued sequence numbers are therefore necessary.

A conversation server following the TRIPLE protocol can be specified as the deterministic finite automaton shown in Figure 4.7. This DFA includes a new "timeout" input as well as the actions "set" and "cancel" for starting and stopping the timer. The server must maintain a sequence number variable as well as a DFA state variable. The sequence number is incremented modulo 3 as part of the DFA action "Complete". The suffices "-1" and "$+1$" appended to message types indicate messages with sequence numbers that do not match the server's current sequence number. Figure 4.7 does not show transitions that simply accept and ignore old duplicate messages.

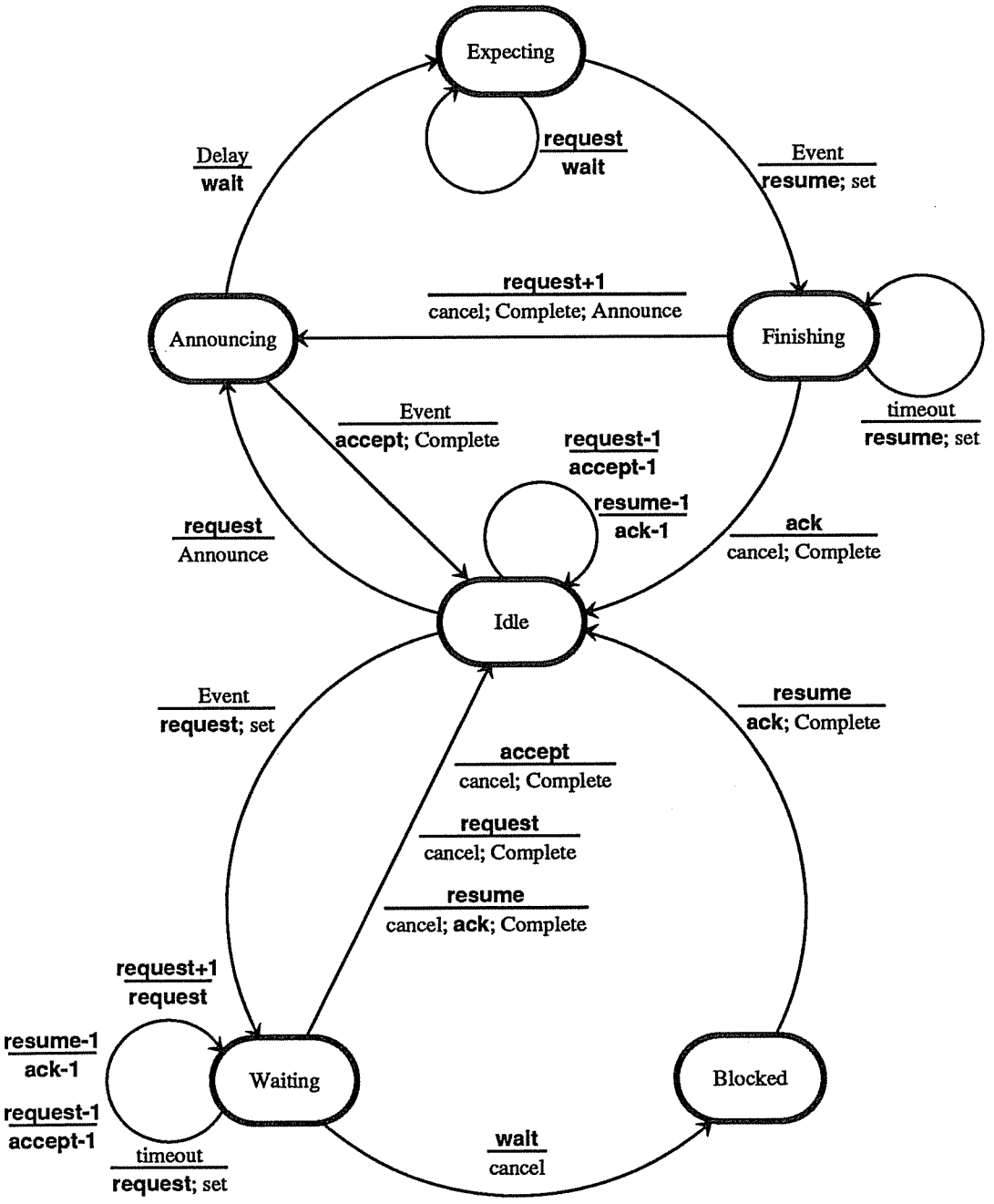

Figure 4.7: The TRIPLE protocol DFA

4.3 Handling Errors

Any implementation of the programming system described in the last chapter must be concerned with program errors. Errors can arise at any stage of the development and use of a new protocol.

Some errors can be detected at protocol generation time, when a conversation description is being processed. Any violation of the conversation description syntax rules can be caught and reported immediately. Inconsistent use of a message class can also be detected. All the instances of a particular message class must specify the same content types.

Some programming errors will be detected when a program that uses conversations is compiled. Badly formed plan-constructor expressions fall into this category. Use of a message class with an incorrect message content type can also be detected at compile time, because the `Put`⟨*class*⟩, `Get`⟨*class*⟩, and `Trade`⟨*class*⟩ plan constructors specify arguments of the correct types for messages of class ⟨*class*⟩. The quality of compile-time error messages will depend entirely on the language and compiler being used.

The remaining types of errors cannot be detected until an application program is running. These errors can be divided into several classes based on how hard it is to detect them.

Some runtime errors can be detected locally, without regard for the behavior of other clients. Such errors include attempts to call blocking functions such as `Wait` inside action procedures. Ambiguous plans and regular expressions that do not terminate properly can also be detected locally at runtime, as can attempts to violate the message-sequencing specifications of a conversation description.

Other runtime errors arise from the joint behavior of two or more clients. An

error occurs when the clients at opposite ends of a conversation decisively choose different paths through a conversation description. A different error occurs when the client at one end of a conversation closes the conversation while the other client is still trying to use it. In this sort of error, neither client can be said to be at fault. These errors are detected when unexpected messages arrive, or when arriving messages require actions that are inconsistent with the local client's behavior.

It is difficult or impossible to detect some programming errors that may occur. If one client uses an incorrect argument in a call to `NewConversation`, the intended connection may never occur. The two supposedly matching initialization calls will return half open conversations that will never become fully open. Application programs may deadlock without violating any of the requirements of the programming system. These errors result in some subset of the application processes being permanently barred from making forward progress. A distributed deadlock detection algorithm would be required to detect them. The cost of running such an algorithm could perhaps be restricted to those processes involved in the deadlock.

Some mechanism for reporting runtime errors must be provided, but the nature of this mechanism will depend on the underlying language. An implementation could simply halt with an error message whenever a runtime error is encountered. Errors could be handled more elegantly in a language such as MESA [Lampson80] or ADA [Ichbiah79] that supports the establishment of exception handlers and the raising of exceptions. In the absence of such features, many of the plan constructors could return error indications, and the function `Wait` could return an indication of any error that occurs during the execution of plans. This last mechanism is not particularly satisfying, because it leads to application programs that are cluttered with error-handling code.

Chapter 5

Protocol Improvement

The previous chapter discussed some general issues involved in implementing the protocol system defined in chapter 3, but it did not consider ways to use available application-specific information to improve the efficiency of the application's communication. There are two sources of such information: plans and conversation descriptions. Each of these sources can be used to enhance communication efficiency.

In general, we will attempt to reduce the total number of messages that must be sent. All networks have a per-message cost of transmission as well as a per-byte cost, so reducing the number of messages sent can be an improvement, even if the total number of transmitted bytes is constant.

5.1 Plan-based Improvement

A plan built by a client may contain information not only about the immediate communication requirements of the client, but also about the client's future communication requirements. The client's agent is often able to use this additional knowledge to package the client's communication in fewer messages than would be

possible if the knowledge were not available.

For example, consider the participants in a remote procedure call conversation (see section 3.1, page 14). First suppose the customer accesses the server by making the following calls:

```
Perform (PutQuestion (Server, Question));
Perform (GetAnswer (Server, Answer));
```

In this case the customer's agent sees the two operations one at a time, so it must accomplish them individually. At best each operation will require an exchange of low-level messages, so the entire remote procedure call will require at least four low-level messages.

Instead, suppose the client makes the following call:

```
Perform (
    Sequence (
        PutQuestion (Server, Question),
        GetAnswer (Server, Answer)));
```

The customer's agent can now see both operations at once, so it can send a single low-level message that describes both of them. Then the server's agent can respond with a single low-level message completing both operations, and the entire remote procedure call will have been accomplished with two low-level messages.

This example involves a straight sequence of operations, but the principle can be extended to more complicated plans. Suppose the remote procedure call conversation description allowed the reply to a question to be either an answer or an error, and that the customer makes the following call:

```
Perform (
    Sequence (
        PutQuestion (Server, Question),
        Choice (
            GetAnswer (Server, Answer),
            GetError (Server, Error))));
```

The customer's agent can send a single message describing the entire plan, and the server's agent can respond with a message completing the question operation and one of the alternative replies.

In general, an agent interpreting a decisively specified event in a plan's equivalent DFA can include in its outgoing message the connected region of the DFA rooted at the event and involving the same conversation as the event. The message must include the states and transitions of the DFA region and the communication events that label the transitions, including any outgoing client data specified by those events. The region can be discovered by a depth-first search starting at the decisive DFA state and terminating at any state that has an exit transition labeled with either an action procedure or a communication event belonging to a conversation other than the decisively specified event's conversation. If the region includes some of the exit transitions of a DFA state, it must include all the transitions. Two other criteria may also limit the size of the region that can be included in an outgoing message. First, the packet size of the underlying communication system may limit the number of communication events that can be included in any one message. Second, the message cannot include any event whose outgoing data might be affected by the incoming data of a predecessor event.

When an agent receives a message containing a plan fragment, it can accomplish events in the fragment locally until it reaches a state in the fragment that has no exit transitions. Then it responds with a message describing the events it completed, so that the original agent can complete the same events.

If fragment-containing messages cross because both events of a matching pair are specified decisively, then each agent can compare the fragment it sent with the fragment it received in order to complete a maximal sequence of events. The comparison is not computationally expensive, because one fragment or the other

must be decisive at each state. The fragments might not describe plan regions of equal size. If part of one of the fragments remains unaccomplished after a maximal sequence of events has been completed, both agents can regard the excess as a new fragment that has already been transmitted.

The computational overhead of searching through plans to decide what parts to include in an outgoing message is significant. However, a regular expression can be used many times, either because it is part of a cycle or because the client reuses it, so the overhead can be amortized over all the uses by preprocessing the expression the first time it is encountered. The preprocessing step decorates each event in a regular expression with a description of the relevant part of the remainder of the plan. When an agent encounters a decisively specified event, it can include in its outgoing message the description stored at the event.

The algorithm described above assumes an underlying reliable communication service. In fact, the EXCHANGE protocol described in section 4.2.1 is a special case of this algorithm that always transmits plan fragments consisting of a single communication event.

We can describe the new algorithm more concretely if we assume, as in the last chapter, that an agent consists of a collection of abstract plan drivers and conversation servers. The interface between plan drivers and conversation servers must be slightly modified. Plan drivers can now give their conversation servers entire plan regions to be accomplished, not just single communication events. A driver that encounters a decisively specified communication event during the execution of its plan gives the server the entire relevant region rooted at the event. Announcements generated by a server still describe a single decisively specified remote event, but the driver's response to an announcement can be an entire region rooted at the local event that matches the announced remote event. A server now notifies

its driver of a region's completion only when an entire path through the region has been accomplished. The notification must include the last DFA state on the completed path.

Conversation servers can then execute the algorithm of Figures 5.1-5.2 within the framework provided by the modified server-driver interface.

The plan-packaging strategy described in this section can also be integrated into the TRIPLE protocol described in section 4.2.2. We now allow **request** messages to describe entire plan regions rather than single communication events. A **request** message's matching **accept** or **resume** message then describes a completed path through the region described in the **request** message.

A serious complication arises because **request** messages can cross on the communication medium. If the **request** messages describe plan regions that match in length on the common path, both agents may complete the events on that path and proceed. The problem arises when one message describes a longer path than the other. In this case both agents can complete the short common path. Then the agent that received the longer region must treat the excess as a new **request** message with a new sequence number, and the agent that sent the longer region must pretend it sent the new **request**. Unfortunately, finite automata cannot count, so these requirements cannot be easily integrated into the DFA of Figure 4.7 that defines the TRIPLE protocol. An auxiliary counter is needed, and some of the DFA transitions require enabling predicates.

```
procedure ConversationServer (Conversation).
var LocalPlan, RemotePlan.
begin
    while (Conversation is open) do
        Await a new RemotePlan from our partner or
                a new LocalPlan from our plan driver.
        if (we got a new RemotePlan) then
            LocalMatch (RemotePlan).
        else {we got a new LocalPlan}
            RemoteMatch (LocalPlan).
        end
    end
end ConversationServer.

procedure LocalMatch (RemotePlan).
var CompletedLocalPlan, LocalPlan, Event.
begin
    Clear CompletedLocalPlan.
    while (RemotePlan is not exhausted) do
        if (RemotePlan is decisive) then
            Announce the decisively specified event to our plan driver.
        end
        Await a new LocalPlan from our plan driver.
        foreach Event in Path (LocalPlan, RemotePlan) do
            Accomplish Event.
            Append Event to CompletedLocalPath.
        end
        if (LocalPlan is exhausted) then
            Notify our plan driver of LocalPlan's completion.
        end
    end
    Send CompletedLocalPath to our partner.
    if (LocalPlan is not exhausted) then
        RemoteMatch (LocalPlan).
    end
end LocalMatch.
```

Figure 5.1: A conversation server's algorithm

```
procedure RemoteMatch (LocalPlan).
var RemotePlan, Event.
begin
    Send LocalPlan to our partner.
    while (LocalPlan is not exhausted) do
        Await a new RemotePlan from our partner.
        foreach Event in Path (LocalPlan, RemotePlan) do
            Accomplish Event.
        end
    end
    Notify our plan driver of LocalPlan's completion.
    if (RemotePlan is not exhausted) then
        LocalMatch (RemotePlan).
    end
end RemoteMatch.

function Path (var LocalPlan, var RemotePlan) : list of Event.
var EventList.
begin
    Clear EventList.
    while (LocalPlan and RemotePlan are not exhausted) do
        if (LocalPlan is decisive) then
            Append the decisively specified local event to EventList.
            Locate the matching event in RemotePlan.
        else {RemotePlan must be decisive}
            Locate the matching event in LocalPlan.
            Append the matching local event to EventList.
        end
        Advance the states of LocalPlan and RemotePlan.
    end
    returnEventList.
end Path.
```

Figure 5.2: A conversation server's algorithm – continued

5.2 Conversation-based Improvement

The second major source of application-specific information is the set of conversation descriptions that specify the form of the application's communication. The types and sizes of application messages and possible message orders are directly available from the descriptions. Information about how the conversations are actually used cannot be determined until runtime. Such information includes the distribution of choices made at alternatives in a conversation, the time intervals between messages, and whether a particular message is sent or received decisively. To collect such information, the system must keep statistics while the application is running. With enough accurate information, agents can sometimes improve the efficiency of the application's communication by delaying certain low-level messages in hopes of packaging them with succeeding messages or even omitting them entirely.

5.2.1 Statistics Collection

Statistics about how a conversation is used are most easily maintained if the conversation description is stored as a deterministic finite automaton (DFA). Techniques for transforming an arbitrary regular expression into a DFA are well known [Aho77]. Applying these techniques to the regular expression of a conversation description results in a DFA that consists of a set of states connected by transitions that are labeled with primitive elements of the regular expression. In our case, the primitive elements describe message transmissions, so the transitions are labeled with ordered pairs, each consisting of a message class and a message direction chosen from {-->, <--, <->}. The agents at both ends of the conversation can use the same DFA, but for the first participant of the conversation the label ($\langle class \rangle$, -->)

describes a Put⟨*class*⟩ operation while for the second participant the same label describes a Get⟨*class*⟩ operation. A similar statement holds for the "<--" message direction.

When an agent completes a communication event for its client, it can match the event against the transitions leaving the current conversation state. Since the automaton is deterministic, exactly one of the transitions will match the event, and the agent can change the state of the conversation to that specified by the matching transition. The event *causes* the transition.

To make the protocol improvement decisions discussed in the next section an agent must have some knowledge of the future course of the conversation. In particular, it needs the following information about each of the next several transitions.

- What transition will be taken next.
- Whether the next transition will be preceded by a communication event involving a different conversation.
- Whether the event that causes the transition will be specified decisively.
- When the client will provide the event that causes the transition.
- When the client will wait for the event's completion.

To estimate this information, it is natural to attach attributes to each transition of the DFA and to adjust their values each time the transition is taken. The following attributes are useful.

Likelihood. In general, a state in the DFA will have several transitions leaving it, and each time the conversation reaches that state it must follow exactly one of the exit transitions. The choice of which transition to take is made by the client driving the conversation, or by the client at the other end if the local client is not decisive. The *Likelihood* of a given transition is the estimated probability that the transition will be taken once the conversation reaches the DFA state from which the transition originates. Some DFA states will have only one exit transition, in which case the single transition will have a *Likelihood* of 1.

Continuity. Since plans may involve several conversations, the next transition in a particular conversation may not be made until one or more events involving other conversations are completed. The *Continuity* of a transition is the estimated probability that the transition follows the previous transition without any intervening events involving other conversations.

Decisiveness. The events that cause transitions in the conversation DFA can be specified decisively or indecisively. The *Decisiveness* of a transition is the estimated probability that an event that causes the transition will be specified decisively. A transition can have a *Decisiveness* less than 1 even if it is the only transition leaving a particular DFA state, because plans can involve several conversations.

Delay. Clients perform some amount of work in addition to producing communication events. The *Delay* of a transition is the estimated amount of time the client spends working before providing an event that causes the transition and after providing the previous event.

Overlap. Once a communication event has been accomplished, it is returned to the client, but the client may do some work before actually waiting for the event. The *Overlap* of a transition is the estimated amount of work the client does after providing an event that causes the transition and before waiting for that event to be completed.

The *Delay* and *Overlap* attributes are estimates of time intervals, the distribution of which can be characterized in many different ways. We have chosen to maintain a minimum and maximum estimated interval. See section 7.5 for some further ideas on this subject.

These attributes should accurately characterize the client's recent behavior, so the values assigned to a transition's attributes should depend only on measurements made during the k most recent uses of that transition, where k is a small integer parameter. If k is too small, short-term fluctuations in the client's behavior may cause the attribute values to change too rapidly and to never accurately reflect the client's future behavior. If k is too large, long-term changes in the client's behavior may not be reflected in the transition attributes for an unacceptably long time.

For a given k we can define the attribute values as follows: A transition's *Likelihood* is the fraction of the last k times the transition was enabled that it was actually taken. A transition is enabled whenever its conversation reaches the DFA state from which the transition originates. The *Continuity* of a transition is the fraction of the last k times the transition was taken that it immediately followed another transition in the same conversation. The *Decisiveness* of a transition is the fraction of the last k events causing the transition that were specified decisively. A transition's *Delay* is the range of the amount of work the client did before providing each of the last k events that caused the transition, and its *Overlap* is the range of

the amount of work the client did after providing but before awaiting each of those events. Each attribute value is a function of the last k observations of some variable. *Likelihood*, *Continuity*, and *Decisiveness* are averages of the last k observations of Boolean variables, while *Delay* and *Overlap* are ranges of the last k observations of non-negative real variables.

Unfortunately, attributes defined in this manner cannot be maintained cheaply. To maintain an average over the last k observations of a variable it is necessary to remember the k most recent observations. A new observation can be incorporated into the average in constant time by subtracting a k^{th} of the oldest value from the average and adding a k^{th} of the new value. The new value is then remembered in place of the oldest value. To maintain the minimum and maximum of the last k observations of a real variable, it is also necessary to remember the k most recent observations, but in this case it is not possible to incorporate a new observation in constant time. When a new value is incorporated, it is easy to check if it is a new minimum or maximum. The problem arises when the displaced value is equal to the minimum or maximum of the observations. If so, the minimum or maximum of the new set must be recalculated. The recalculation can be done in time proportional to $\log k$ using a heap.

A simple and inexpensive alternative to remembering many observations and maintaining complicated data structures is to clear a transition's attributes and start over every k^{th} time the transition is taken. The disadvantage of this approach is that sometimes a transition's attributes will be based on as many as k observations while at others they will be based on only one observation. A somewhat more complicated alternative is to keep and use an old set of attributes based on k observations while the next set is being constructed. Then the attributes used to make decisions will always be based on k observations, but they will never

incorporate the most recent observations. To overcome this disadvantage at some cost in time, decisions can be based on both the saved set of attributes and the set under construction.

5.2.2 Opportunities for Improvement

Information about how a conversation is being conducted can sometimes be used to improve the conversation's efficiency. In general, an opportunity for packaging messages into one packet arises whenever two or more consecutive low-level messages belonging to a single conversation are sent in the same direction. An improvement attempt often involves delaying a particular outgoing message in the hope that it can be packaged with a succeeding message or omitted entirely. Such delays can degrade the conversation's performance if the hoped-for succeeding message is not available in time. Information gathered at runtime can be used to avoid this degradation. The underlying protocol used to implement plans and conversations determines the particular improvements that are possible. The remainder of this chapter discusses the opportunities that arise in the TRIPLE protocol described in section 4.2.2.

Delaying **wait** messages

A straightforward improvement is to delay a **wait** message in hopes of sending an **accept** message instead. According to the basic protocol, an agent **B** that has received a **request** message from agent **A** but whose client has not yet submitted a matching event responds to the **request** with a **wait** message. Instead, **B** can delay the **wait** for a short amount of time. If **B**'s client produces the matching event during that time, the **request** message can be answered with an **accept** message, completing the transaction in just two messages (Figure 5.3a). Otherwise, **B** must

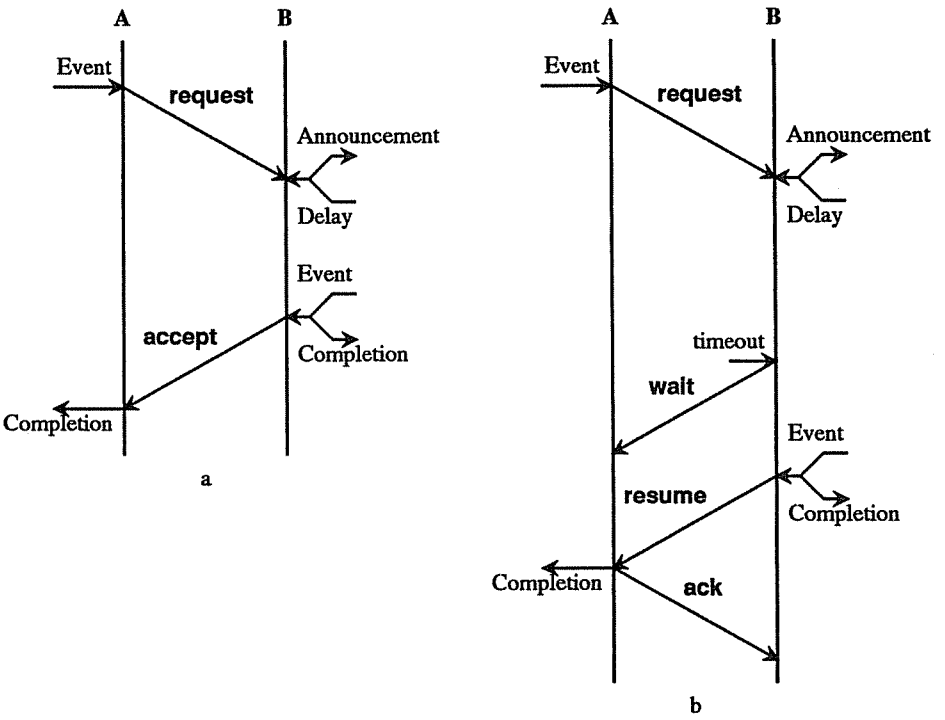

Figure 5.3: Delaying a **wait** message

still send the **wait** message, and the transaction will eventually be completed with **resume** and **ack** messages (Figure 5.3b).

Runtime estimates of the client's behavior can be used to decide how long a particular **wait** message should be delayed. Agent **B** knows from the incoming **request** message which transition the conversation DFA will next take, and the *Delay* attribute of that transition estimates the minimum and maximum time it takes the client to produce events that cause that transition. An appropriate **wait** delay is therefore the maximum *Delay* minus any client time that elapsed before the **request** message arrived. Agent **B** can set a timer to expire if the awaited communication event does not arrive during the expected interval, in which case the **wait** message must be returned by itself. Agent **A** must know how long **B** is willing to delay the **wait** message so it can allow time for both a message round trip and **B**'s **wait** delay before retransmitting its original **request** message.

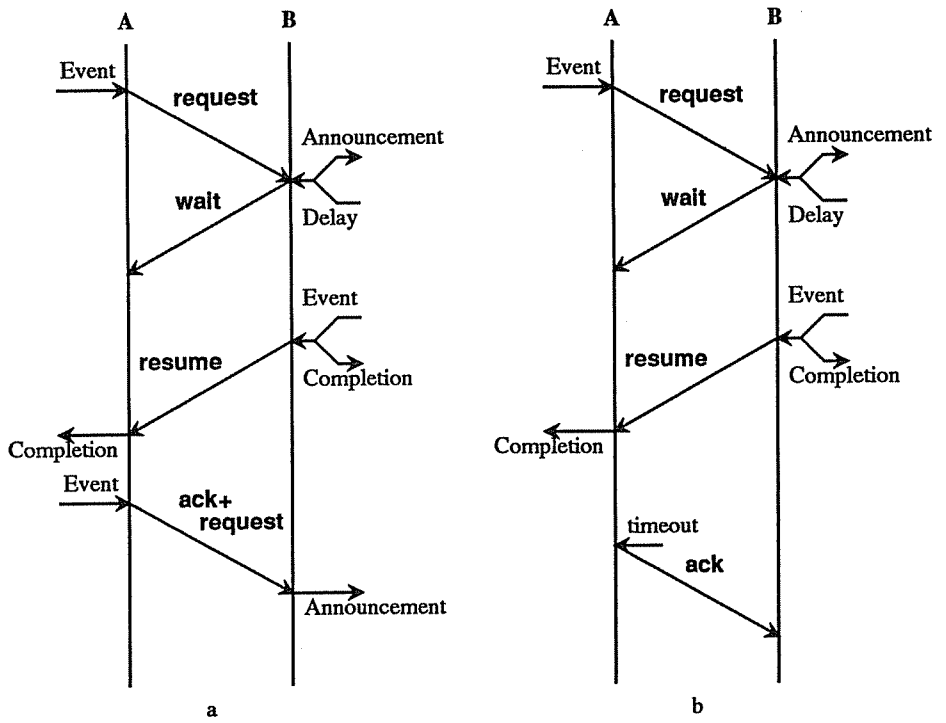

Figure 5.4: Delaying an **ack** message

Choosing a bad **wait** delay time has several consequences. If the delay is not large enough, no improvement occurs, and in fact the agent incurs the additional costs of setting a timer and having it expire. If it is larger than necessary, an unnecessarily long time is spent recovering from lost **request**, **wait**, or **accept** messages, because A's **request** retransmission time depends on B's **wait** delay time. In fact, there may be some *a priori* upper bound imposed on **wait** delays to limit the time it takes to recover from lost messages. If the appropriate delay exceeds this bound, the **wait** message should be returned immediately to avoid the expense of setting a timer.

Delaying ack messages

A second possible improvement is to delay an **ack** message in hopes of including it in the next outgoing **request** message. For example, suppose agent A has sent

a **request** message to agent **B**, received a **wait** message in response, and sometime later received a **resume** message. **A** must respond to the **resume** with an **ack** message. If the next communication event is specified decisively and is already available, **A** may package the **ack** and the succeeding **request** in a single message. If the next event is not yet available, **A** may delay the **ack** for a short amount of time. If **A**'s client submits the next communication event during that time, **A** can send the **ack** and the succeeding **request** together (Figure 5.4a). If the event is still not available at the end of that time, **A** must send the **ack** message by itself (Figure 5.4b).

Once again, estimates of the client's behavior can be used to select an appropriate **ack** delay. While it is delaying the **ack**, agent **A** cannot know which of several alternative transitions the conversation will next take, so it must rely on the *Likelihood* attributes of the alternatives to guess at the future course of the conversation. The agent can restrict its attention to those alternatives with *Likelihood* attributes significantly greater than zero.

An **ack** message should not be delayed if the *Continuity* of a likely next transition is significantly less than one at either end of the conversation. A *Continuity* less than one indicates a client that may be involved in other conversations before it gets around to continuing the current conversation, in which case the **ack** should be returned immediately to avoid conflicts with other conversations.

Agent **A** should not delay the **ack** message unless the *Decisiveness* of all the likely next transitions is close to one. If **A**'s next communication event is not specified decisively, **A** cannot initiate the next transaction with a **request** message. Agent **B** cannot initiate the next transaction until it receives the **ack** to its outstanding **resume** message, so **A** must return the **ack** immediately.

Another situation in which the **ack** should be returned immediately occurs

when the clients of both **A** and **B** specify the next communication event decisively, but **B**'s client provides the event sooner than **A**'s client. In this case **A** should return the **ack** immediately so that **B** can initiate the next transaction as soon as possible. This situation can be detected by comparing the *Delay* attributes of the likely transitions with the corresponding attributes at the other end of the conversation. Agent **A** should not delay the **ack** if **A**'s maximum *Delay* attribute of a likely transition is greater than **B**'s minimum *Delay* of the same transition.

If all the criteria discussed above are satisfied, delaying the **ack** message will probably be worthwhile. An appropriate delay time is the largest of the maximum *Delay* attributes of all the likely next transitions. Delaying the **ack** by that amount of time should allow **A**'s client enough time to provide the next communication event no matter which of the likely next transitions is actually taken. Agent **A** must set a timer that will expire if the awaited event does not arrive as quickly as expected, so that the **ack** message can be returned by itself. Agent **B** must allow time for both a message round trip and **A**'s **ack** delay before retransmitting the **resume** message, and for that reason there may be some *a priori* upper bound on **ack** delays. If the appropriate delay exceeds that upper bound, the **ack** should be returned immediately to avoid the expense of setting a timer.

Delaying accept messages

A third possible improvement is to delay an **accept** message in hopes of packaging it with the next outgoing **request** message. Suppose agent **B** has received a **request** message from agent **A** and is about to send an **accept** message in response. If **B**'s next communication event is available and is specified decisively, **B** can package the **accept** and the succeeding **request** in a single message. Otherwise **B** may delay the **accept** message for a short amount of time. If **B**'s client submits the next

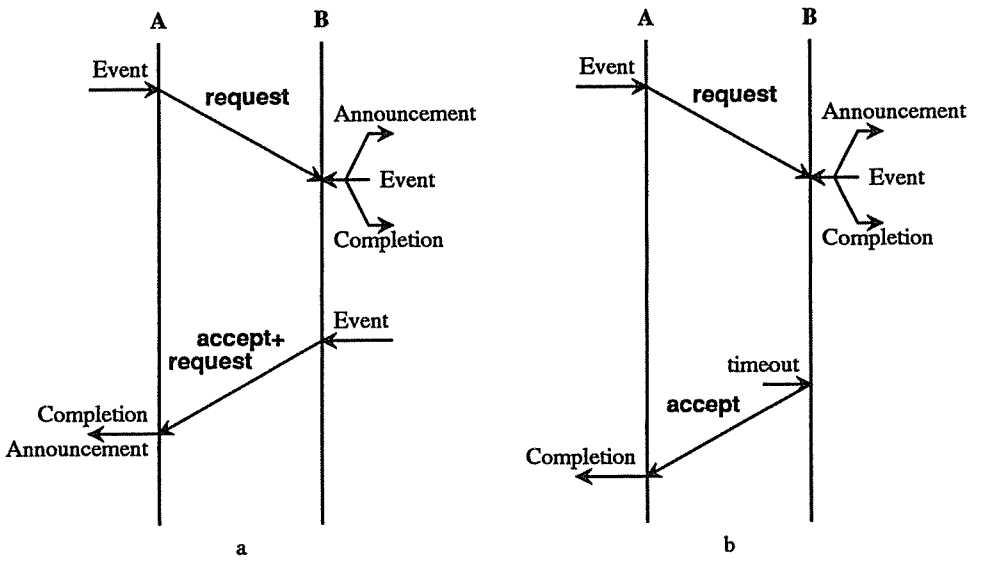

Figure 5.5: Delaying an **accept** message

communication event during that time, **B** can send the **accept** and the following **request** together (Figure 5.5a). Otherwise the **accept** must still be sent by itself (Figure 5.5b).

Several considerations that affect the choice of a proper **ack** delay also apply to the choice of an appropriate **accept** delay. The *Likelihood* attributes of the conversation's possible next transitions can be used to predict the future course of the conversation. The **accept** message should not be delayed when the *Continuity* attributes of the likely next transitions show that other conversations may interfere. The *Decisiveness* attributes of the likely next transitions can be used to avoid delaying the **accept** when the awaited communication event may not be specified decisively. If this improvement is attempted, the largest of the maximum *Delay* attributes of the likely transitions is an appropriate setting for the timer used to detect events that are not submitted as quickly as expected. In this case the sender of the original **request** message must allow time for a message round trip, a **wait** delay, and an **accept** delay before retransmitting the **request**, because both delays may occur in succession. The **accept** should not be delayed at all if the appropriate

delay exceeds some *a priori* bound.

An additional complication arises because both **accept** and **request** messages may carry client data. The delayed **accept** message and the succeeding **request** message may not both fit in a packet of the underlying transmission medium. In that case the **accept** and **request** must be sent as separate messages, so the **accept** message may as well be sent immediately.

Once the criteria discussed so far have been met, an agent considering the delay of an **accept** message must still decide if the attempted improvement will in fact be profitable. Delaying an **accept** message can sometimes serialize client work that could otherwise be done in parallel.

Consider the sequence of two transactions illustrated in Figures 5.6 and 5.7. The vertical lines in these figures represent time in application processes **A** and **B**. The heavy solid segments of these time lines indicate intervals of client computation, and the dotted segments indicate idle intervals. Gaps in the lines indicate intervals of agent activity. Client computations are occasionally interrupted by incoming messages. The labels on the time lines indicate client actions such as initiating computations, appending events to plans, or waiting for events to complete.

In Figure 5.6, agent **A**'s client appends a communication event and awaits its completion before beginning to work on the second event. If agent **B** returns the first **accept** message immediately, the clients can work on their next events in parallel (Figure 5.6a). If, however, agent **B** delays the **accept** while its client works on the next event, **A**'s client will not be able to begin work on the second event until **B**'s client is finished (Figure 5.6b). The savings realized by decreasing the number of messages is overwhelmed by the decrease in client parallelism.

In Figure 5.7, agent **A**'s client works on the second communication event before awaiting the completion of the first event. Agent **B** can now safely delay the **accept**,

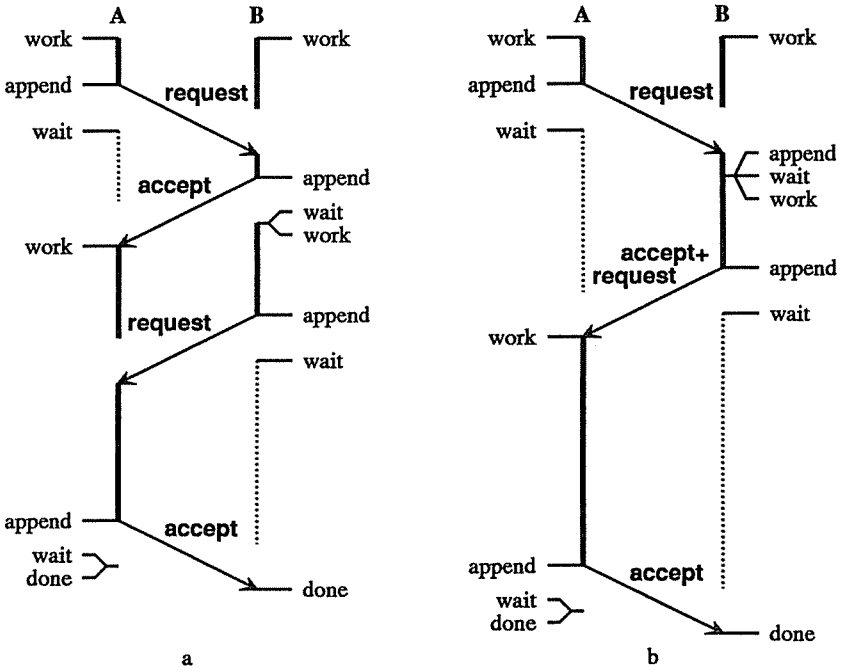

Figure 5.6: An unsuccessful **accept** delay

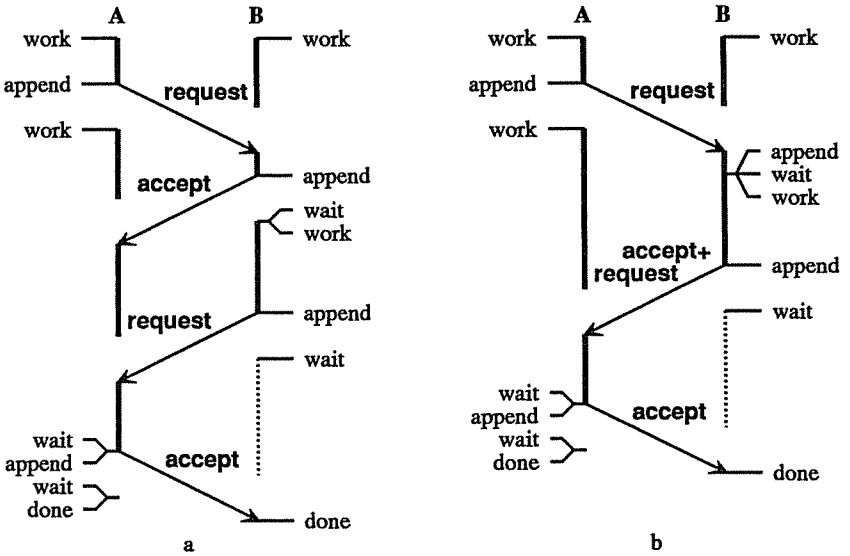

Figure 5.7: A successful **accept** delay

and the saving of a message is beneficial.

Agents can use the *Overlap* attributes of transitions to distinguish these two situations. If agent **A** has sent a **request** message, the *Overlap* attribute of **A**'s current transition indicates how much work **A**'s client will probably do before waiting for the current transaction to complete. If the *Delay* attributes of all of **B**'s likely next transitions are smaller than **A**'s current *Overlap*, **B** can probably delay the **accept** long enough to return a combined **accept** and **request** message before **A**'s client needs the **accept**.

The algorithm for deciding whether to delay an **accept** message is further complicated by the fact that the attempted improvement may be beneficial in the long run even if it delays one of the clients temporarily. The delayed client may be compensated by a decreased waiting time in some future transaction. Before delaying an **accept** message, an agent must decide that some future transaction will be completed by both agents sooner than it would be if the improvement were not attempted. In general, the transaction at which the benefit becomes apparent may be arbitrarily far in the future, but in most cases the benefit becomes apparent in the next one or two transactions, and it is possible for the agent to recognize those situations.

Consider the sequence of three transactions in Figure 5.8. In the first transaction agent **A** sends a **request** message, and **A**'s client immediately waits for the transaction's completion. When agent **B** is ready to complete the first transaction with an **accept** message, it must decide whether delaying the **accept** will benefit both **A** and **B** over the course of the three transactions. If **B** delays the **accept**, **A**'s client will be blocked waiting for the first transaction to complete while **B**'s client is busy working on the second transaction. This disparity is acceptable if two conditions are met. First, **B**'s client must be able to work on the third transaction

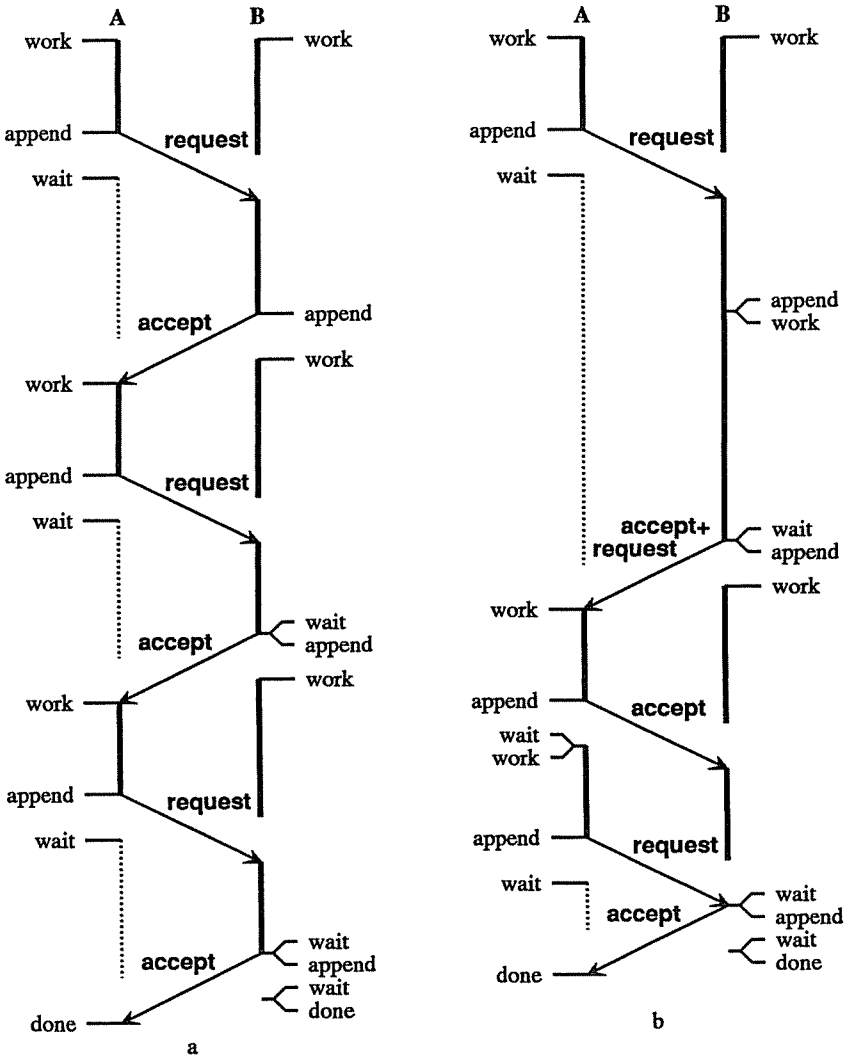

Figure 5.8: Another successful **accept** delay

while the second transaction is completing, because otherwise working ahead on the second transaction was of no benefit. Second, **A**'s client must be able to do the work for both the second and third transactions while **B**'s client is working on the third transaction, because otherwise **A**'s client cannot make up the time it was delayed on the second transaction. The first condition amounts to the statement that **B**'s *Overlap* attribute of the transition corresponding to the second transaction must exceed **A**'s *Delay* for the second transaction. The second condition is equivalent to the statement that the sum of **A**'s *Delay* attributes for the second and third transactions must be less than **B**'s *Delay* for the third transaction. These statements can be relaxed somewhat by allowing for **A**'s first transaction *Overlap*, which measures the amount of work **A** can do while **B** is delaying the **accept** message. These two conditions are sufficient for **B**'s delay of the **accept** message to be beneficial to both agents. Figure 5.8 illustrates a scenario in which both conditions hold, and agent **B**'s delay of the first **accept** message is in fact profitable. In practice, agent **B** cannot always know the next two transitions the conversation will take, and the conditions are more complicated if the likely future course of the conversation has branches.

5.2.3 The LAZY Protocol

The conversation-based improvement attempts discussed in the last section can be incorporated in the TRIPLE protocol DFA of Figure 4.7 by adding new DFA states to represent the time intervals during which messages are being delayed and new message types to account for combined messages. We call the modified protocol the LAZY protocol because to at least some extent it only sends messages when necessary. Figure 5.9 illustrates the modified DFA that defines the LAZY protocol. This illustration does not show transitions that correspond to expired

retransmission timers and duplicate messages, and it does not include actions that start and stop retransmission timers or generate completion notices. The complete set of DFA transitions is listed in Figures 5.10-5.11.

The decision algorithms discussed in the last section are encapsulated in the DFA actions "set-wt", "set-ak", and "set-ac", which are outlined in Figures 5.12-5.14. A decision not to delay a particular message is implemented by scheduling an immediate timeout, in effect bypassing the message delay state. These three procedures depend on the variable CurrentState that holds the state of the conversation DFA. This variable is modified as events are completed. Each transition of the conversation DFA has "NextState" and "MsgSize" attributes as well as the statistical attributes defined in section 5.2.1. The decision algorithms depend on the values of statistics at both ends of the conversation. Subscripts "local" and "remote" on transition attributes are used to indicate which values are intended. Superscripts "$\perp$" and "$\top$" are used on transition attributes that hold ranges of real values to indicate the lower or upper bound of the range.

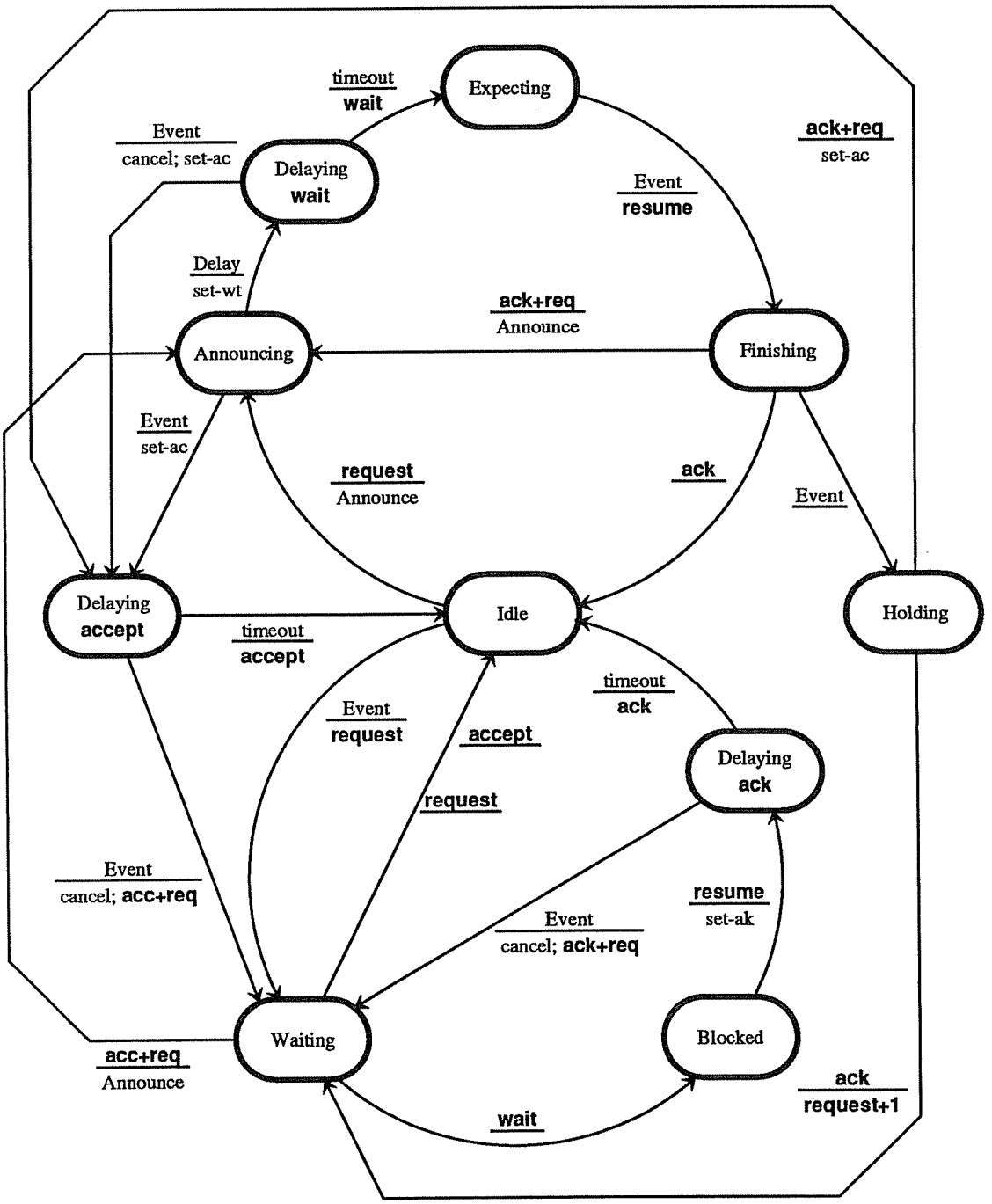

Figure 5.9: The LAZY protocol DFA

State	Input		Next State	Action
Idle				
	Event	$\Longrightarrow$	Waiting	: **request**; set
	request	$\Longrightarrow$	Announcing	: Announce
	request−1	$\Longrightarrow$	Idle	: **accept**−1
	resume−1	$\Longrightarrow$	Idle	: **ack**−1
	accept−1	$\Longrightarrow$	Idle	:
	ack−1	$\Longrightarrow$	Idle	:
	accept−2	$\Longrightarrow$	Idle	:
	ack−2	$\Longrightarrow$	Idle	:
Announcing				
	Event	$\Longrightarrow$	DelayAc	: Complete; set-ac
	Delay	$\Longrightarrow$	DelayWt	: set-wt
DelayWt				
	Event	$\Longrightarrow$	DelayAc	: cancel; Complete; set-ac
	timeout	$\Longrightarrow$	Expecting	: **wait**
	request	$\Longrightarrow$	DelayWt	:
	accept−1	$\Longrightarrow$	DelayWt	:
	ack−1	$\Longrightarrow$	DelayWt	:
Expecting				
	Event	$\Longrightarrow$	Finishing	: **resume**; Complete; set
	request	$\Longrightarrow$	Expecting	: **wait**
	accept−1	$\Longrightarrow$	Expecting	:
	ack−1	$\Longrightarrow$	Expecting	:
Finishing				
	ack	$\Longrightarrow$	Idle+1	: cancel
	ack+**req**	$\Longrightarrow$	Announcing+1	: cancel; Announce
	request+1	$\Longrightarrow$	Announcing+1	: cancel; Announce
	Event	$\Longrightarrow$	Holding	:
	timeout	$\Longrightarrow$	Finishing	: **resume**; set
	request	$\Longrightarrow$	Finishing	:
	accept−1	$\Longrightarrow$	Finishing	:
	ack−1	$\Longrightarrow$	Finishing	:

Figure 5.10: The LAZY Protocol DFA transitions

State	Input		Next State	Action
Holding				
	ack	$\Longrightarrow$	Waiting+1	: cancel; **request**+1; set
	ack+req	$\Longrightarrow$	DelayAc+1	: cancel; Complete; set-ac
	request+1	$\Longrightarrow$	DelayAc+1	: cancel; Complete; set-ac
	timeout	$\Longrightarrow$	Holding	: **resume**; set
	request	$\Longrightarrow$	Holding	:
	accept−1	$\Longrightarrow$	Holding	:
	ack−1	$\Longrightarrow$	Holding	:
DelayAc				
	Event	$\Longrightarrow$	Waiting+1	: cancel; **acc+req**; set
	timeout	$\Longrightarrow$	Idle+1	: **accept**
	request	$\Longrightarrow$	DelayAc	:
	accept−1	$\Longrightarrow$	DelayAc	:
	ack−1	$\Longrightarrow$	DelayAc	:
Waiting				
	request	$\Longrightarrow$	Idle+1	: cancel; Complete
	accept	$\Longrightarrow$	Idle+1	: cancel; Complete
	acc+req	$\Longrightarrow$	Announcing+1	: cancel; Complete; Announce
	wait	$\Longrightarrow$	Blocked	: cancel
	resume	$\Longrightarrow$	DelayAk	: cancel; Complete; set-ak
	timeout	$\Longrightarrow$	Waiting	: **request**; set
	request−1	$\Longrightarrow$	Waiting	: **accept**−1
	resume−1	$\Longrightarrow$	Waiting	: **ack**−1
	request+1	$\Longrightarrow$	Waiting	: **request**
	accept−1	$\Longrightarrow$	Waiting	:
	ack−1	$\Longrightarrow$	Waiting	:
	accept−2	$\Longrightarrow$	Waiting	:
	ack−2	$\Longrightarrow$	Waiting	:
Blocked				
	resume	$\Longrightarrow$	DelayAk	: Complete; set-ak
	wait	$\Longrightarrow$	Blocked	:
DelayAk				
	Event	$\Longrightarrow$	Waiting+1	: cancel; **ack+req**; set
	timeout	$\Longrightarrow$	Idle+1	: **ack**
	resume	$\Longrightarrow$	DelayAk	:

Figure 5.11: The LAZY Protocol DFA transitions – continued

procedure set-wt.
    $t \leftarrow$ the transition from CurrentState determined
            by the most recently received **request.**
    $delay \leftarrow t.Delay^{\top}_{\text{local}} -$ (time since last event completion).
    SetTimer ($delay$).
end set-wt.

procedure SetTimer ($interval$).
    if $interval >$ MaxMessageDelay **then**
        Schedule an immediate timeout.
    else
        Schedule a timeout to occur when $interval$ has elapsed.
    end
end SetTimer.

Figure 5.12: The **wait**-delay decision algorithm

procedure set-ak.
    $delay \leftarrow 0$.
    foreach Likely transition t from CurrentState **do**
        $delay \leftarrow \max\ (delay, t.Delay^{\top}_{\text{local}})$.
        if Disconnected **or** Indecisive **or** Harmful **then**
            Schedule an immediate timeout.
            return.
        end
    end
    SetTimer ($delay$).
end set-ak.

Likely $\equiv (t.Likelihood > \epsilon)$

Disconnected $\equiv (t.Continuity_{\text{local}} < (1 - \epsilon))$

Indecisive $\equiv (t.Decisiveness_{\text{local}} < (1 - \epsilon))$

Harmful $\equiv (t.Delay^{\top}_{\text{local}} > t.Delay^{\bot}_{\text{remote}})$

Figure 5.13: The **ack**-delay decision algorithm

```
procedure set-ac.
    msgsize ← (size of the outgoing accept).
    delay ← 0.
    foreach Likely transition t from CurrentState do
        delay ← max (delay, t.Delay^⊤_local)
        overlap ← t.Overlap^⊥_remote − (time since last request arrived).
        remainder ← t.Delay^⊤_remote − overlap.
        if Disconnected or Indecisive or TooBig or
                            (HarmfulNow and HarmfulLater) then
            Schedule an immediate timeout.
            return.
        end
    end
    SetTimer (delay).
end set-ac.
```

Likely $\equiv (t.Likelihood > \epsilon)$

Disconnected $\equiv (t.Continuity_{\text{local}} < (1 - \epsilon))$

Indecisive $\equiv (t.Decisiveness_{\text{local}} < (1 - \epsilon))$

TooBig $\equiv (msgsize + t.\text{MsgSize} > \text{MaxPacketSize})$

HarmfulNow $\equiv (t.Delay^{\top}_{\text{local}} > overlap)$

HarmfulLater $\equiv$ ($\exists$ a transition u from t.NextState such that:
$(u.Likelihood > \epsilon)$ and (
$(u.Overlap^{\bot}_{\text{local}} < remainder)$ or
$(u.Delay^{\bot}_{\text{local}} < u.Delay^{\top}_{\text{remote}} + remainder)$))

Figure 5.14: The **accept**-delay decision algorithm

5.2.4 Observations

The discussion in the preceding sections suggests several issues that need to be clarified.

Statistical Significance

Several of the transition attributes defined in section 5.2.1 are estimated probabilities, but in practice the actual estimates are never used. All that is important is whether an attribute is significantly greater than zero and whether it is significantly less than one. Since each estimated probability is an average of k observations of a Boolean variable and k is a small integer, even one positive observation among the last k may make the estimate significantly greater than zero and one negative observation may make it significantly less than one. Agents are interested not so much in the average of the k observations but in whether the average is zero, one, or neither, and this three-valued attribute may be easier to maintain and use.

Sharing Statistics

Some of the protocol improvement strategies discussed above require knowledge of the behavior of the client at the other end of the conversation as well as of the behavior of the local client. Therefore an agent must maintain two sets of attributes, one that characterizes the local client and one that characterizes the remote client. Agents must exchange sets of attributes describing their clients, since an agent can only measure its own client. This exchange can be accomplished by including transition attributes in the low-level messages that are used to complete the transition. Under some of the statistics collection schemes discussed in section 5.2.1, the local attribute values are only brought up to date every k^{th} time the transition is taken,

so they need to be exchanged only at those times.

Making Decisions

The protocol improvement strategies discussed in previous sections can be complicated and computationally expensive. However, the attributes on which the decisions are based change only occasionally, so the decisions themselves need to be reconsidered only at those times. The decision values can then be stored with the attributes on which they are based and used when they are needed.

Both agents involved in a transaction must know about improvement attempts that involve the delay of messages, because retransmission intervals must allow for possible message delays. Since both agents have the same information, they can make the same decisions independently, or one agent can make the decisions and report them to the other agent.

Using Timers

Timers are used solely to recover from errors. In the past they were used to recover from hardware errors such as lost messages. Now we also use them to recover from bad decisions. Ideally, timers should never expire, and for that reason setting and canceling a timer should be inexpensive, but the actual delivery of a timeout need not be efficient.

Chapter 6

Concrete Implementation

A prototype implementation we call **ProGen** has been constructed to demonstrate the feasibility of the protocol system design and of some of the protocol improvement strategies outlined in the last chapter. This prototype supports application processes written in Modula-2 [Wirth82] and intended for the Crystal multicomputer [Cook83]. The Crystal multicomputer is a collection of 20 Digital Equipment Corporation VAX 11/750's connected by an 80 megabit/second Proteon token ring. Each node machine runs a copy of the Crystal *Nugget*, which provides software partitioning of the network and which supports both datagram and reliable communication services within a partition.

The ProGen system has two major components, the protocol generator itself and the ProGen runtime environment, which is maintained as a library that can be linked into application programs. Figure 6.1 diagrams the steps required to construct a Crystal load module.

Conversation descriptions are processed by the protocol generator, which produces a Modula-2 module for each description. These modules are compiled along with the application processes written in Modula-2. The resulting relocatable ob-

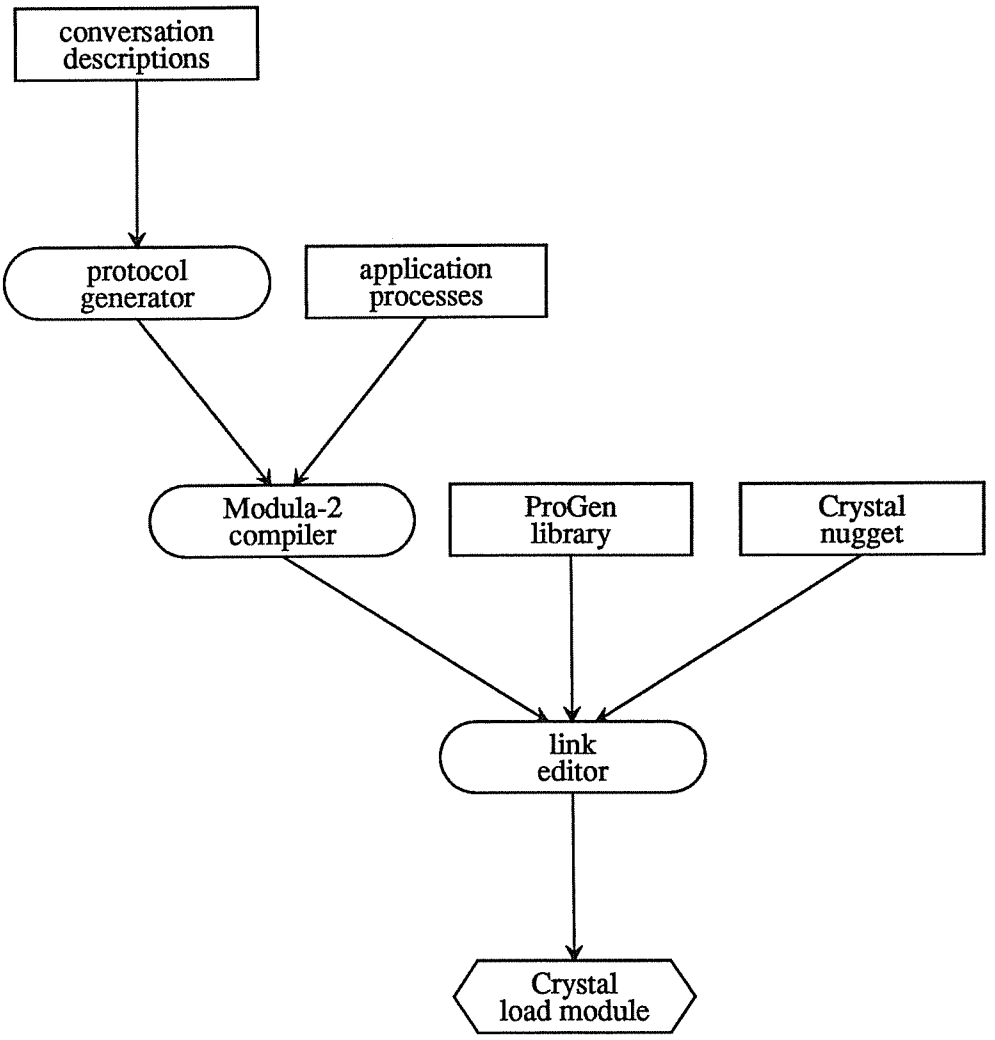

Figure 6.1: Construction of a Crystal load module

jects are linked with the ProGen library and the Crystal Nugget to form load modules that can be executed in a Crystal partition. This implementation allows only a single application process per Crystal node machine.

We will begin with a description of the ProGen runtime environment, because it supports the generated protocols. Then we will discuss the protocol generator, several versions of which have been implemented so that various protocol improvement strategies can be fairly evaluated.

6.1 The ProGen Runtime Environment

The ProGen library provides a runtime environment for application processes that use conversations. It defines all the data types required by application processes, and it contains all the application-level functions and procedures described in chapter 3 that are not specific to individual conversations.

The environment's lowest layer is a task-scheduling subsystem, and the environment is structured as a collection of lightweight tasks communicating through input queues. The application process is itself embodied in a task, while another task executes action procedures on the client's behalf. A third task coordinates the opening and closing of conversations. Tasks associated with each open conversation implement conversation-specific protocols, and tasks associated with each active plan coordinate plan access. Figure 6.2 illustrates the logical organization of a program running on a Crystal node machine. We will begin with a description of the task subsystem, followed by a description of the tasks that comprise the environment.

6.1.1 Task Subsystem

The Crystal nugget supports a single user process per Crystal node machine. The task subsystem provided by the ProGen environment supports lightweight tasks within a nugget process.

A new task may be created dynamically by naming a Modula-2 procedure to serve as the task's template and specifying a stack size and priority for the task. Task stacks are allocated from a heap. A task terminates when it returns from the procedure that forms its template. New tasks are given internal names when they are created.

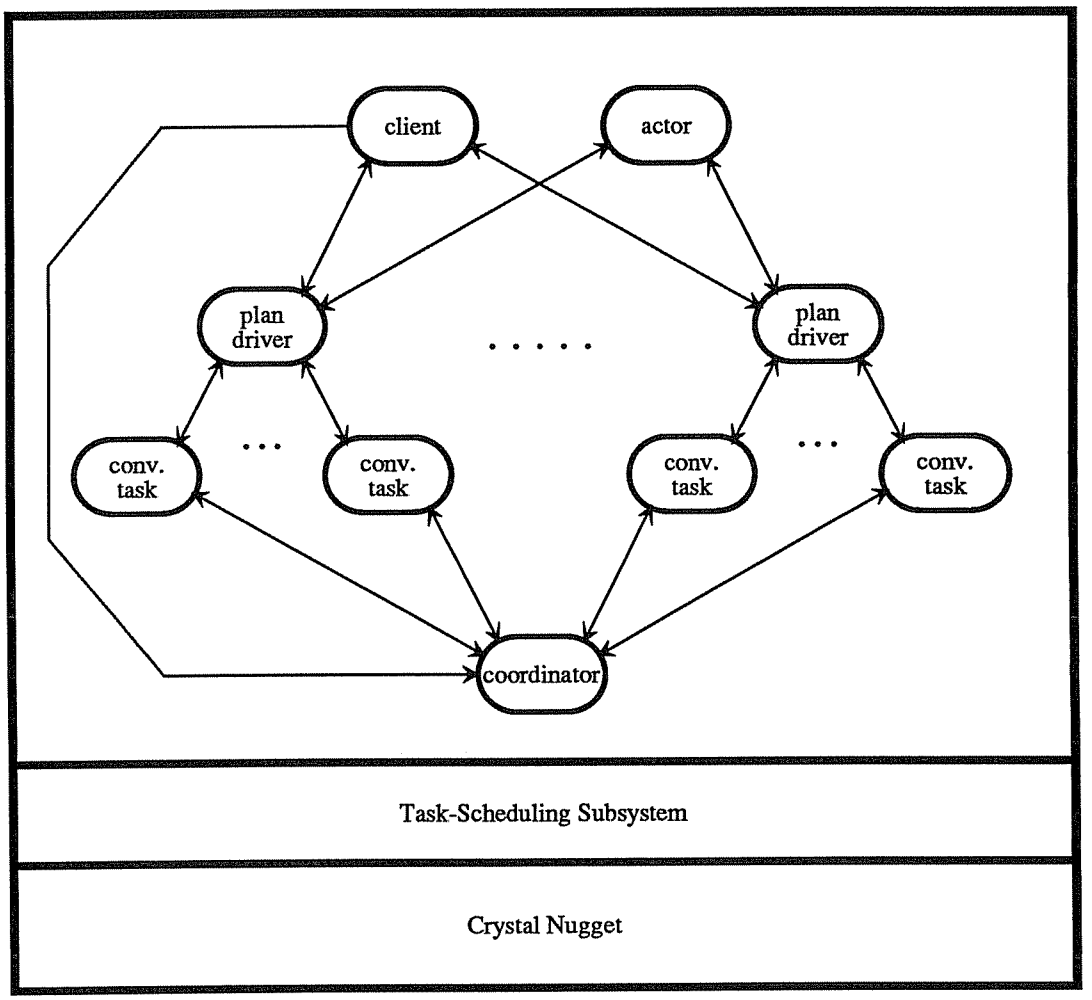

Figure 6.2: Logical organization of a Crystal node machine

Tasks are scheduled according to their priorities. A task may be preempted at any time by a task of higher priority, but tasks within a priority level may rely on mutual exclusion. A single "idle" task has priority lower than all other tasks.

Tasks communicate by enqueueing "notices" to each other. A notice can be any data object so long as it allows room for the necessary queueing pointers. By convention, notices have an initial field that identifies the data type of the remainder of the notice. Tasks have a single input queue from which notices may be dequeued. A task attempting a dequeue when its queue is empty will be blocked, allowing other tasks of equal or lower priority to run. Enqueueing a notice to a task of higher priority will result in the immediate scheduling of that task, but the preempted task will maintain its position with respect to its peers.

The task subsystem converts interrupts from the Crystal nugget into notices enqueued to appropriate tasks. The Crystal nugget provides a number of communication channels called "ports". The ProGen environment associates a task with each port, and send or receive completion interrupts that occur on a port are directed to the associated task. The task subsystem also provides an alarm mechanism that allows a task to request that a "timeout" notice be enqueued to the task at a certain time in the future. An alarm may be canceled by the task that set it.

6.1.2 Client and Actor Tasks

Application programmers never manipulate tasks directly. The main body of an application process is incorporated in a "client" task, which runs at a priority above that of the idle task. The routines a client calls to manipulate conversations and plans actually enqueue requests to other system tasks that run at a still higher priority. The client `Wait` routine is essentially a dequeue from the client's input

queue.

A task called the "actor" executes action procedures as they are encountered in the plans constructed by the client task. The actor and the client run at the same priority, so action procedures are not executed asynchronously with respect to the client.

6.1.3 Conversation Coordinator

Conversations are managed by a task called the "coordinator". Client calls to `NewConversation` and `CloseConversation` generate requests to the coordinator. Coordinators on different machines use a well known nugget port to communicate among themselves. They follow the algorithm outlined in section 4.1 to discover matching pairs of conversation initialization requests. A nugget port is allocated to each new conversation. The coordinator runs at a priority higher than that of the client.

6.1.4 Conversation Servers

Associated with each open conversation is a "conversation server" created by the coordinator when the conversation is initialized. The procedure template from which a conversation server is created depends on the particular conversation description being used. The protocol generator described later produces a customized conversation server template for each different conversation description.

The servers at opposite ends of a conversation are responsible for accomplishing the data transfers that comprise the conversation. They communicate with each other using the nugget port allocated to the conversation. The protocol they maintain on this channel is determined by the protocol generator, and it is this protocol we have in mind when we speak of "generating" a protocol for a particular

conversation description. Conversation servers run at a high priority so that other tasks cannot unduly interfere with the communication protocol.

6.1.5 Plan Drivers

Each active plan has an associated "driver" task created when the client calls `NewPlan`. New regular expressions that are appended to the plan are enqueued to the plan driver, which then cooperates with the actor and with the conversation servers associated with the plan in interpreting the expressions. Regular expressions are enqueued back to the client as they are completed. The driver is also responsible for closing the plan cleanly when the client calls `ClosePlan`.

Plan drivers and conversation servers cooperate according to the server-driver interface described in section 4.2. Plan drivers enqueue decisively specified events to appropriate conversation servers and servers enqueue announcements and completion notices to their plan drivers.

6.1.6 Overhead Reduction

The task cooperation described in section 4.2 requires many queued notices, and correspondingly many task switches. In our implementation, we try to avoid some of this overhead by allowing conversation servers associated with a particular plan to assume some of the function of the plan driver.

The ProGen library provides a procedure `Advance` that conversation servers may call instead of enqueueing completion notices to the plan driver. The procedure duplicates the operation of the plan driver upon reception of a completion notice, possibly enqueueing a notice to the actor or to another conversation server and possibly returning a regular expression to the client.

The library also provides a function `MatchingEvent` that conversation servers may call instead of enqueueing announcements to the plan driver. The arguments to this function describe a remotely initiated communication event. The function returns a matching event if one exists among the plan's current alternatives. Otherwise it returns `nil` but leaves behind a description of the outstanding announcement.

Reducing task switching overhead in this manner requires that conversation servers associated with a particular plan share a data structure that was previously private to the plan driver. Therefore conversation servers and plan drivers must run at the same priority.

6.2 Protocol Generator

The protocol generator produces conversation server templates tailored to specific conversation descriptions. It also produces the conversation-specific plan constructors that clients use to build communication events.

The protocol generator accepts as input a single conversation description conforming to the syntax of Figure 3.1. As it processes the description it records the conversation name and the names of the participants, as well as all the conversation's message classes and their associated content types. It also parses the regular expression that defines the possible message orderings, and converts it to a minimal deterministic finite automaton using the sequence of algorithms found in [Aho77]. As output, the protocol generator produces a Modula-2 definition module and a corresponding implementation module.

The definition module provides declarations for the conversation-specific constants, variables, and procedures needed by application programs that want to use

conversations based on the given conversation description. The conversation name and the participant names from the header of the conversation description are declared so that they may be used in calls to `NewConversation`. The conversation name points to a data structure that contains information about the conversation description that is needed when a new conversation is created. This information includes the character string name of the conversation description, which the conversation coordinator uses to find matching calls to `NewConversation`. It also includes a pointer to the template procedure from which conversation servers for this conversation description are to be created. The definition module also declares the `Put`⟨*class*⟩, `Get`⟨*class*⟩, and `Trade`⟨*class*⟩ plan constructors that are specific to this conversation description.

The major component of the implementation module produced by the protocol generator is the conversation-specific template for conversation servers. Servers built from this template must cooperate with other conversation servers and with plan drivers in interpreting plans, and they must accomplish the data transfers specified by the communication events directed to them. Conversation servers are also expected to enforce the restrictions on message orders imposed by the conversation description. Any protocol may be used between the servers at opposite ends of a conversation. Different versions of the protocol generator produce servers that use different kinds of protocols.

The implementation module also initializes the variables declared in the definition module and provides procedures for the plan-constructor declarations. The plan constructors are simple functions. The ProGen library defines a data structure that is used to represent the regular expressions that comprise plans. Each plan constructor dynamically allocates a node of this structure and initializes its fields from the constructor's arguments.

We have implemented three versions of the protocol generator that make no attempt to use application-specific information to improve communication efficiency and one version that attempts some of the improvement strategies described in chapter 5. The first three versions serve as benchmarks against which various attempts at improvement can be compared.

All four versions produce conversation servers that use the DFA derived from the conversation description to enforce the description's message ordering restrictions. The DFA is stored as a state transition table, and each conversation server maintains a DFA state variable. As the server begins each new transaction, it determines a new DFA state based on the current state and on the transaction's message class and direction. The transaction is legal if the new state is not an error state. Otherwise the server prints an appropriate error message and aborts the entire application process. Conversation servers produced by the first three versions of the protocol generator make no other use of application-specific information.

6.2.1 EXCHANGE Protocol (Nugget)

The simplest version of the protocol generator creates conversation servers that implement the EXCHANGE protocol defined in section 4.2.1 using the reliable message service provided by the Crystal nugget. The servers accomplish each application-level data transfer by exchanging a pair of reliable nugget messages.

In this version, all the messy details of sequence numbers and retransmission timers are hidden in the nugget, allowing conversation servers that are clean and simple. The nugget has direct access to the hardware clock and to the communication device, so any protocol implemented inside the nugget is bound to be more efficient than the same protocol implemented on top of the nugget, using the timer and the unreliable message services provided by the nugget.

6.2.2 EXCHANGE Protocol (Alternating-Bit)

In another version, conversation servers use the low-level facilities of the nugget to implement a standard alternating-bit protocol (ABP). This intermediate-level protocol is then used to implement the EXCHANGE protocol.

This version can serve as a fair basis for comparison because it depends on the same nugget facilities as all later versions. The conversation servers constructed by this version are still fairly simple because the reliable protocol is logically separate from the algorithm that uses it.

6.2.3 TRIPLE Protocol

The third version of the protocol generator creates conversation servers that use the low-level facilities of the nugget to implement the TRIPLE protocol described in section 4.2.2.

In this version, the EXCHANGE protocol is integrated with the underlying protocol used to achieve reliable communication. This integration allows this version to use fewer low-level messages per transaction than the first two versions, but the constructed conversation servers are more complex.

To help manage this complexity, conversation servers are structured as table-driven finite automata. The heart of each server is a loop with a dequeue operation at its head. Each dequeued notice is converted to a DFA input, which is used along with the current DFA state to locate an entry in the DFA table. The entry contains a new DFA state and a list of actions to be performed. The actions are represented as small constants used to select clauses of a case statement.

The table basically represents the DFA of Figure 4.7 that defines the TRIPLE protocol, but with one modification. The new DFA incorporates a static version

of the protocol improvement strategy described in section 5.2.2, in which a **wait** message is delayed in hopes of sending an **accept** message instead. Conversation servers delay **wait** messages by a fixed length of time, rather than by a length of time based on the past behavior of the client. This fixed delay has two negative consequences. First, the time it takes to recover from lost messages is increased by the length of the delay, and second, the costs of setting a timer and having it expire are incurred if the delay is not long enough. These costs are insignificant compared to the saving of two low-level messages when the strategy is successful.

6.2.4 LAZY Protocol

Our fourth version of the protocol generator creates conversation servers that attempt some of the application-specific protocol improvements discussed in chapter 5. This version implements the LAZY protocol described in section 5.2.3.

Plan-Based Improvements

A simple version of the improvement strategy discussed in section 5.1 has been integrated with the LAZY protocol. This strategy attempts to make use of information contained in the plans provided by clients at runtime.

Conversation servers built by this version of the protocol generator search their associated plans for straight sequences of communication events that can be packaged in a single message. All the events in such a sequence must be part of the same conversation, and the entire sequence, including application data, must fit in a single nugget packet.

When a conversation server is ready to initiate a transaction to accomplish the first event of such a sequence, it can instead initiate a transaction to accomplish the entire sequence. It sends a **request** message describing the entire sequence,

including any outgoing application data specified by the events of the sequence. In reply it expects either an **accept** message that completes the entire sequence or a **wait** message that completes nothing. In the latter case, the transaction will be completed with **resume** and **ack** messages.

A conversation server that receives a **request** message describing a sequence of events responds with an **accept** message if it can match all the events in a short amount of time. Otherwise it responds with a **wait** message and later sends a **resume** message when all the events have been matched.

When crossing **request** messages describe sequences of different length, the conversation servers complete the shorter sequence, and the remainder of the longer sequence is retransmitted in a new **request** message.

The cost of searching for suitable sequences is incurred only when a particular regular expression is encountered for the first time. Therefore clients that reuse expressions or include them in cycles can amortize the cost over all the uses.

Conversation-Based Improvements

Conversation servers built by the fourth version of the protocol generator implement the LAZY protocol, which attempts the conversation-based protocol improvement strategies discussed in section 5.2. These strategies all involve delaying certain low-level messages of the TRIPLE protocol, hoping they can be packaged with succeeding messages or omitted entirely. Under certain circumstances, **wait**, **ack**, or **accept** messages may be delayed.

The algorithm for deciding whether and by how much to delay a particular message requires some knowledge of the past behavior of the client, so conversation servers record relevant characteristics of their clients as they are running. These statistics are attached to the DFA constructed from the conversation description,

which has so far been used only to enforce the message-order constraints of the description. Instead of maintaining attributes for each transition of the DFA, as discussed in section 5.2.1, our implementation maintains attributes for each of the DFA states. This change allows us to use less space, because there are fewer states than transitions, but our characterization of the client is correspondingly less precise, because information that would have been maintained in the exit transitions of a DFA state is now summarized in the state itself. This loss of information could cause us to miss opportunities for protocol improvement, but it will never cause us to attempt an improvement that is not beneficial.

Several of the client attributes maintained by conversation servers involve intervals of time between various client actions such as appending a regular expression to a plan or waiting for the completion of an expression. Such intervals are readily available because the task-scheduling subsystem keeps track of the time consumed by each task, including the client and actor. Unfortunately, time spent by the Crystal nugget servicing a hardware interrupt is attributed to whatever task happens to be running when the interrupt arrives. Clients could be characterized more accurately if this extraneous time could be excluded, but the nugget does not provide the information necessary to make this exclusion possible.

6.3 Performance

We can now discuss the performance of the conversation servers produced by the four versions of the protocol generator. We will first examine two examples for which the protocol improvement strategies discussed in chapter 5 are not applicable. These examples allow us to compare the performance of the two versions of the EXCHANGE protocol with that of the TRIPLE protocol. They also allow us

to assess the cost of attempting protocol improvements in situations where the improvement strategies are not successful. We then examine several examples for which one or another of the improvement strategies are successful.

The protocols we have studied each exhibit several behavior modes. Such modes arise for several reasons. A protocol may use different numbers or types of messages under different circumstances. For example, the TRIPLE protocol uses a **request** message and an **accept** message per transaction when the two clients are reasonably balanced, but it uses **request**, **wait**, **resume**, and **ack** messages when one client is significantly slower than the other. We say the protocol is in **wait-resume** mode when four messages per transaction are required. Different modes also arise from different message crossing patterns. In the ABP version of the EXCHANGE protocol, each transaction consists of exchanged reliable messages. If two machines are well synchronized, each machine's send will complete before its receive completes. If one machine is slightly behind the other, its receive will complete before its send completes. If the machine is further behind, its receive will complete before its send is even initiated. Transaction times may be different in all three cases, even though the numbers and types of messages are the same. Some behavior modes seem to be caused by idiosyncrasies of the communication hardware. Incoming and outgoing messages may interact in strange ways under certain timing conditions. Such modes are hard to explain or predict.

6.3.1 Simple Producer-Consumer Example

Our first example has a simple producer process sending `Item` messages to a simple consumer process. The producer executes the following loop:

```
for i := 1 to Count do
    Produce (Object);
    Perform (PutItem (conv, Object));
end;
```

The consumer executes a similar loop:

```
for i := 1 to Count do
    Perform (GetItem (conv, Object));
    Consume (Object);
end;
```

The time it takes to produce or consume an object can be varied, as can the size of objects.

Figure 6.3 shows the time per iteration as a function of object size using each of the four versions of the protocol generator, assuming it takes no time to produce or consume objects. Object size is varied in 100 byte increments, and each datum is an average over 100 iterations of the loop. The clock is accessed twice, once before and once after the loop, and the difference is divided by the number of loop iterations. The measured interval does not include the opening and closing of conversations, but it does include the early loop iterations when the system may not have achieved a steady state.

A striking feature of this figure is the mode change that occurs in both versions of the EXCHANGE protocol at an object size of about 800 bytes. Apparently the interaction of data messages and acknowledgements changes at that point.

We can draw several conclusions from this figure. First, the behavior of the ABP version of the EXCHANGE protocol is roughly similar to that of the Nugget version, but it is about 4 milliseconds per iteration slower. Protocols implemented on top of the nugget suffer this penalty because they cannot directly access the hardware clock or the communication device.

Second, the TRIPLE protocol significantly outperforms the Nugget version of

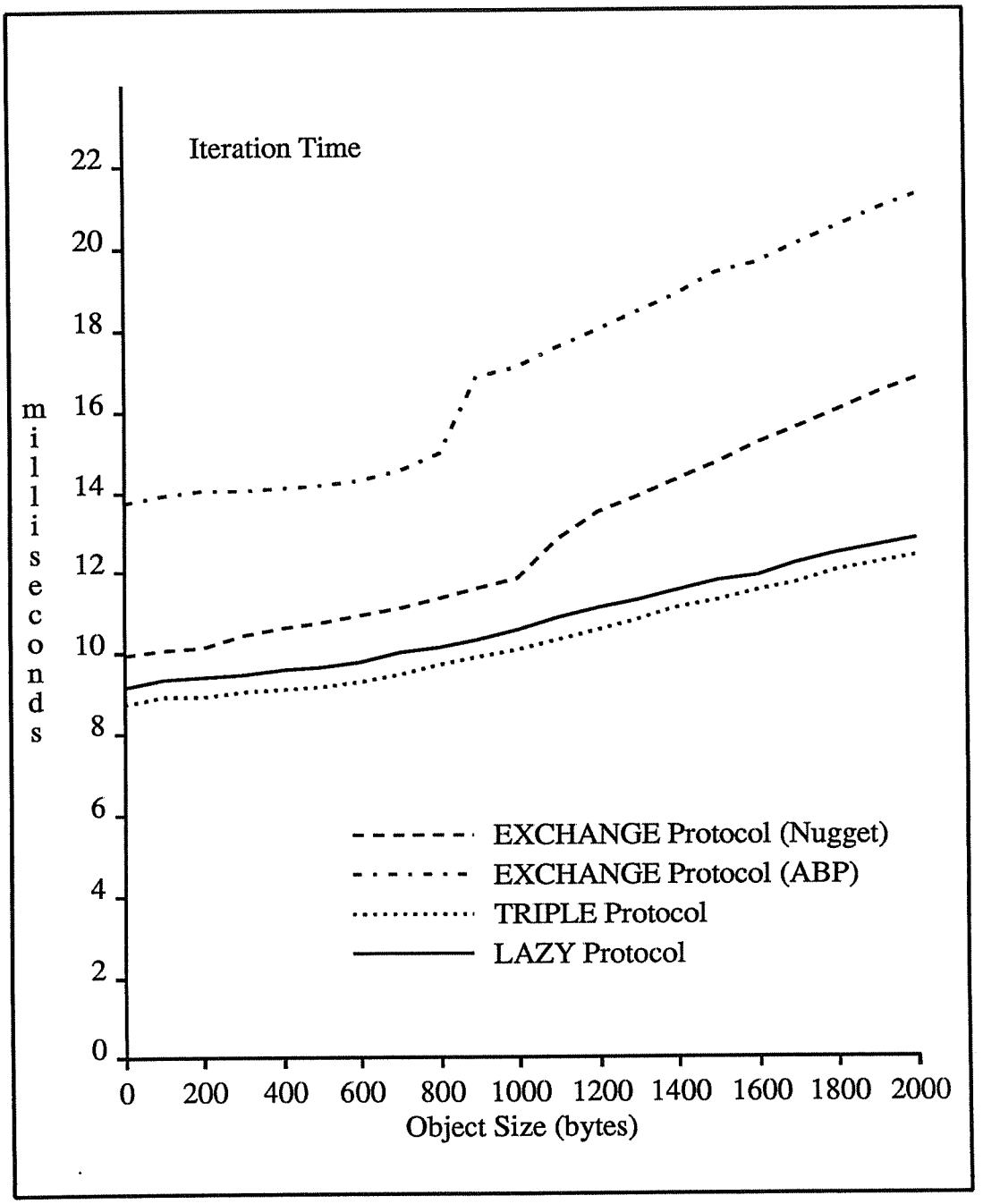

Figure 6.3: Simple producer-consumer – iteration time vs. object size

the EXCHANGE protocol even though it uses the same nugget facilities as the ABP version of the EXCHANGE protocol. The performance advantage would be more pronounced if the TRIPLE protocol had access to the same hardware facilities as the Nugget.

Finally, the LAZY protocol performs only slightly worse than the TRIPLE protocol on which it is based. The difference is the overhead for collecting statistics and looking for improvement opportunities, and we see that it amounts to about 0.7 milliseconds per iteration in this example.

Figure 6.4 illustrates another aspect of this same producer-consumer example. It shows the time per iteration as a function of the time the producer takes to produce an object, assuming objects are four-byte integers and that it takes no time to consume an object. Producer time is varied in 1 millisecond increments, and each datum is an average over 100 loop iterations. The upper set of lines in this figure display the same data as the main graph, but with the producer time subtracted out so that the differences among the versions are more apparent. Several features of this figure must be explained.

First, both versions of the EXCHANGE protocol perform best when the producer and consumer are evenly matched, that is, when the producer takes no time to produce each object. These versions exchange a pair of reliable messages to accomplish each object transmission, and when the clients are evenly matched the two messages cross because both clients are decisive. When the producer is slower than the consumer, the consumer's message arrives before the producer is ready with the next object, so the messages do not cross. Apparently this pattern of messages is less efficient than the crossing pattern. This phenomenon may also explain the mode change in the last figure, in which the producer and consumer were evenly matched but the exchanged messages were not of equal size.

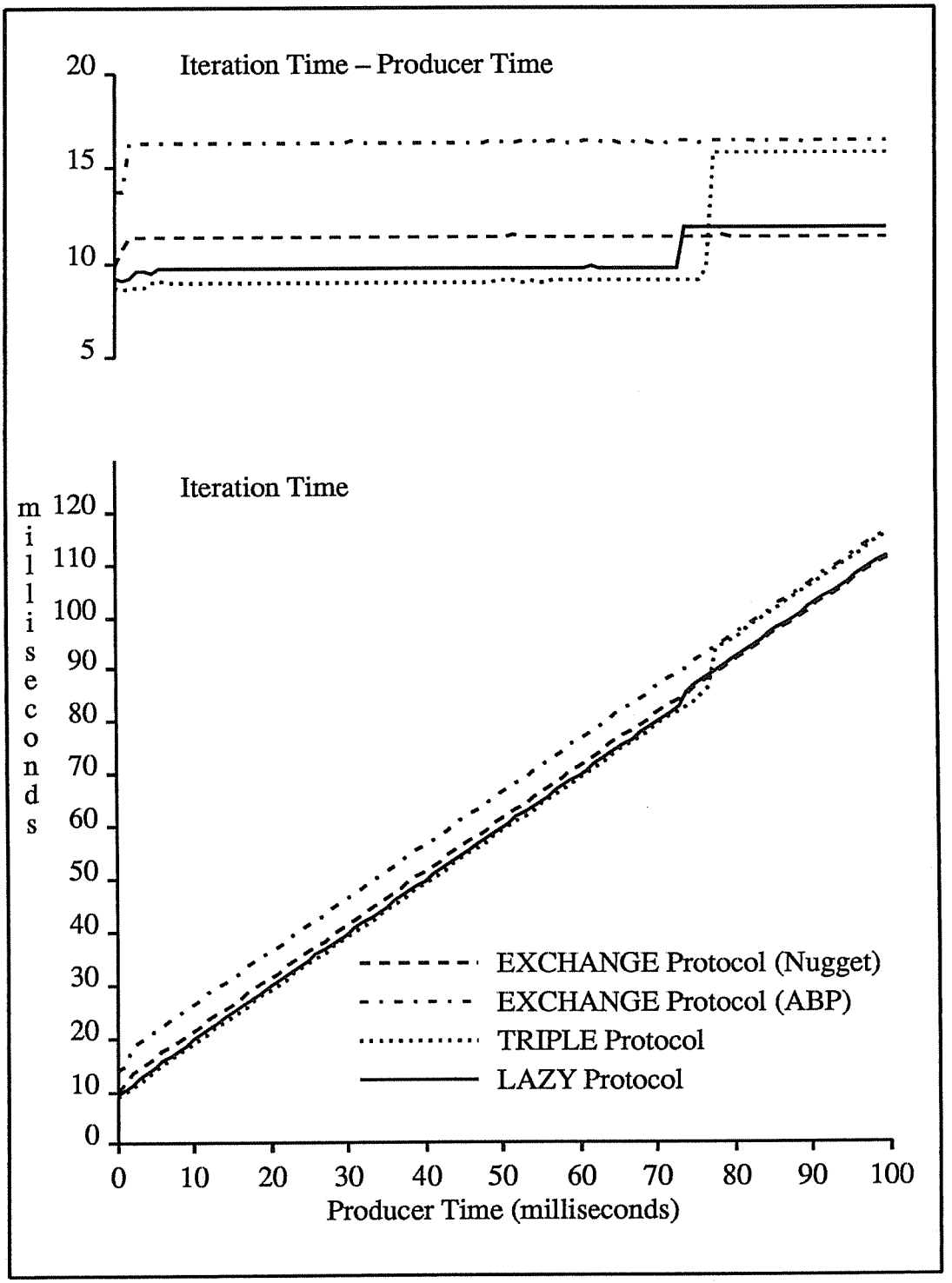

Figure 6.4: Simple producer-consumer – iteration time vs. producer time

A second feature of this figure is the sudden jump in iteration time that occurs at a producer time of 75 milliseconds in the TRIPLE protocol curve. This version of the TRIPLE protocol incorporates a fixed **wait** delay of 75 milliseconds, so this jump marks the transition to the **wait-resume** mode of the protocol. The conversation servers use two messages per iteration below the transition and four messages per iteration above the transition. The ABP version of the EXCHANGE protocol uses four messages per iteration throughout the range, and we see that its performance is only slightly worse than the TRIPLE protocol in **wait-resume** mode.

The LAZY protocol experiences a similar jump in iteration time near the 75 millisecond mark, but the increase is about 3 milliseconds instead of 7. The LAZY protocol does not have a fixed **wait** delay, but it does have an *a priori* upper bound of 75 milliseconds on how long any message may be delayed, so this jump again marks the transition to **wait-resume** mode. In this situation the LAZY protocol does not delay the **wait** message at all, thereby avoiding the costs of setting a timer and having it expire. However, the main difference between the TRIPLE and LAZY protocols in this range is that the LAZY protocol is able to successfully apply one of its protocol improvement strategies. The last of the four messages per iteration in **wait-resume** mode is an **ack** message from the consumer to the producer acknowledging the producer's **resume** message. In this example, the consumer's next **request** message is ready very quickly, so the consumer can profitably delay the **ack** so it can be included in the next **request.** The LAZY protocol therefore uses three messages per iteration instead of four, accounting for most of its 4 millisecond per iteration performance advantage over the TRIPLE protocol.

6.3.2 Simple Remote Procedure Call Example

Our second example involves a remote procedure call conversation between a server and a simple client. The server executes the following statement:

```
Perform (
    Sequence (
        Cycle (
            Sequence (
                GetQuestion (conv, Question),
                Action (ComputeAnswer),
                PutAnswer (conv, Answer))),
        GetExit (conv)));
```

The client executes the following loop:

```
for i := 1 to Count do
    GenerateQuestion;
    Perform (PutQuestion (conv, Question));
    Perform (GetAnswer (conv, Answer));
end;
Perform (PutExit (conv));
```

The time it takes to generate questions and compute answers can be varied arbitrarily. Both questions and answers are four-byte integers. The server's `GetQuestion` event in this example is not specified decisively because the `GetExit` event is always a possible alternative. The `Answer` transaction is specified decisively at both ends.

Figure 6.5 shows the time per iteration for this example as a function of the time it takes the server to compute an answer, assuming the client takes no time to generate questions. Each iteration now represents two high-level messages or as many as eight low-level messages. Server time is varied in 1 millisecond increments, and each datum is an average over 100 loop iterations.

In this figure we again see the transition to **wait-resume** mode of both the TRIPLE and LAZY protocols near a server time of 75 milliseconds. However, only the answer transaction is affected, so even above the transition the protocols use

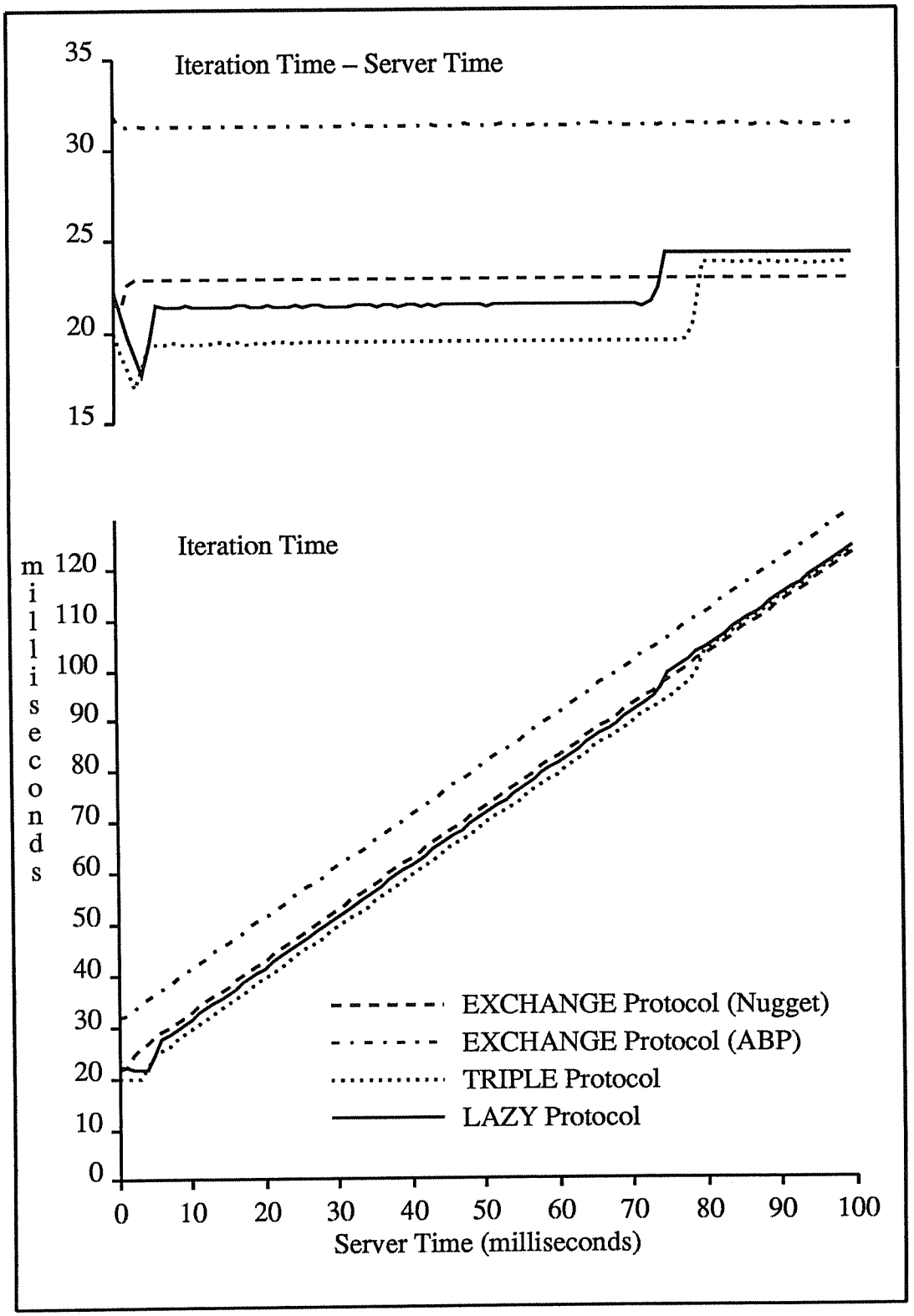

Figure 6.5: Simple remote procedure call – iteration time vs. server time

only six low-level messages per iteration. In this example, the criteria for delaying the **ack** message are not met, so the LAZY protocol is not able to outperform the TRIPLE protocol either above or below the transition to **wait-resume** mode. However, delaying the **ack** message would in fact be profitable in this situation, so we see that our conservative decision algorithms sometimes miss valid improvement opportunities.

The LAZY protocol's overhead for collecting statistics and searching for improvement opportunities is about 2 milliseconds per iteration in this example. Each iteration represents two high-level messages, but even so the overhead is higher in this example than in the last.

The TRIPLE and LAZY protocols exhibit sharp performance peaks near a server time of 3 or 4 milliseconds. The peak marks the point at which the conversation tasks generate **request** messages for the answer transaction at nearly the same time. Below the peak the server initiates the answer transaction, while above the peak the client initiates the transaction.

6.3.3 Packaged Remote Procedure Call Example

In our next example we again have a remote procedure call conversation. The server is unchanged from the last example, but the client is somewhat more sophisticated and executes the following loop:

```
for i := 1 to Count do
    GenerateQuestion;
    Perform (
        Sequence (
            PutQuestion (conv, Question),
            GetAnswer (conv, Answer)));
end;
Perform (PutExit (conv));
```

This client packages the `PutQuestion` and `GetAnswer` events in a single regular expression rather than executing them individually.

Figure 6.6 shows the time per iteration as a function of the server time for the TRIPLE and LAZY protocol versions of the protocol generator. The performance of the two versions of the EXCHANGE protocol on this and later examples is not shown, because we expect it to be consistent with the earlier examples. The same is true of the TRIPLE protocol version, but we continue to show its performance for comparison with the LAZY protocol version. Once again, server time is varied in 1 millisecond increments and each datum is an average over 100 loop iterations.

In this example, the LAZY protocol is able to take advantage of the additional information provided by the client to package both the question and answer events in a single **request** message. The server's conversation task can respond with a single **accept** message accepting the question and providing the answer, completing each iteration with just two low-level messages. However, the overhead for packaging messages is fairly high, so the LAZY protocol outperforms the TRIPLE protocol by only about 15 percent instead of the 50 percent that might be expected.

Above the 75 millisecond mark, the server responds to the double **request** message with an immediate **wait** message and later returns a double **resume** message. The client's **ack** is sent by itself, so each iteration requires four low-level messages in this range. In the same range the TRIPLE protocol requires six messages per iteration, two for the question transaction and four for the answer transaction.

6.3.4 Packaged Producer-Consumer Example

We turn now to a less realistic example but one that shows that taking advantage of the information in plans can be very profitable. In this example, as in the first example, a producer process sends `Item` messages to a consumer process, but

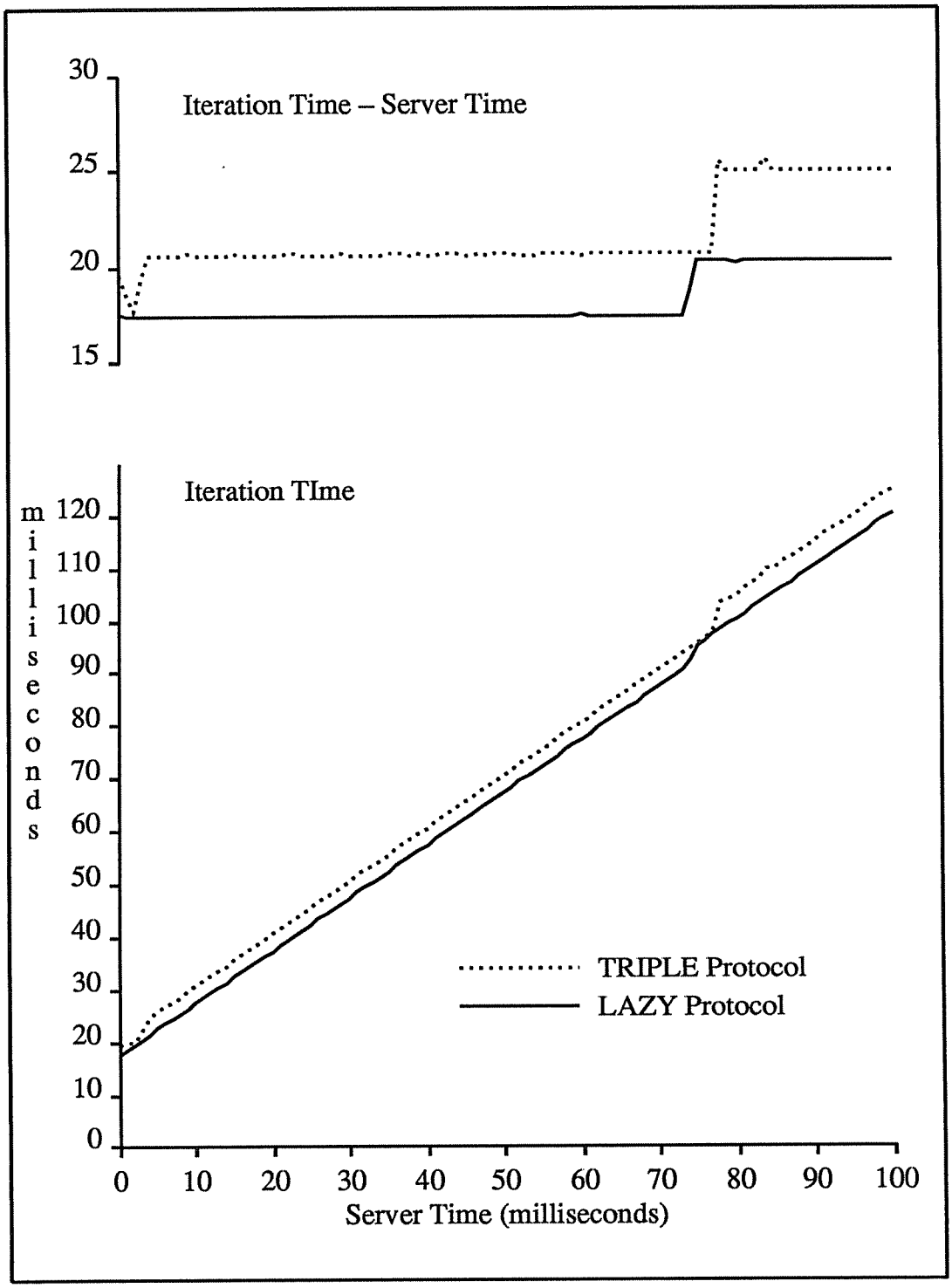

Figure 6.6: Packaged remote procedure call – iteration time vs. server time

in this case the processes package several communication events in each regular expression. The producer executes the following code fragment:

```
putexpr := nil;
for i := 1 to Package do
    putexpr := Sequence (putexpr, PutItem (conv, Object[i]));
end;
for i := 1 to Count div Package do
    Produce (Object[1 .. Package]);
    Append (putexpr, plan);
    Wait ();
end;
FreeExpression (putexpr);
```

The consumer executes a corresponding code fragment:

```
getexpr := nil;
for i := 1 to Package do
    getexpr := Sequence (getexpr, GetItem (conv, Object[i]));
end;
for i := 1 to Count div Package do
    Append (getexpr, plan);
    Wait ();
    Consume (Object[1 .. Package]);
end;
FreeExpression (getexpr);
```

In this example we can vary the time it takes to produce and consume objects, we can vary the size of objects, and we can vary the number of objects that are packaged in each regular expression. The loop limits are adjusted so that tests with different package counts generate the same number of transactions. We will be concerned with the time per transaction rather than the time per iteration of the loop.

Figure 6.7 shows the time per transaction as a function of the number of events per package, using four-byte objects and assuming it takes no time to produce or consume objects. Package count is varied from 1 to 10, and each datum is an

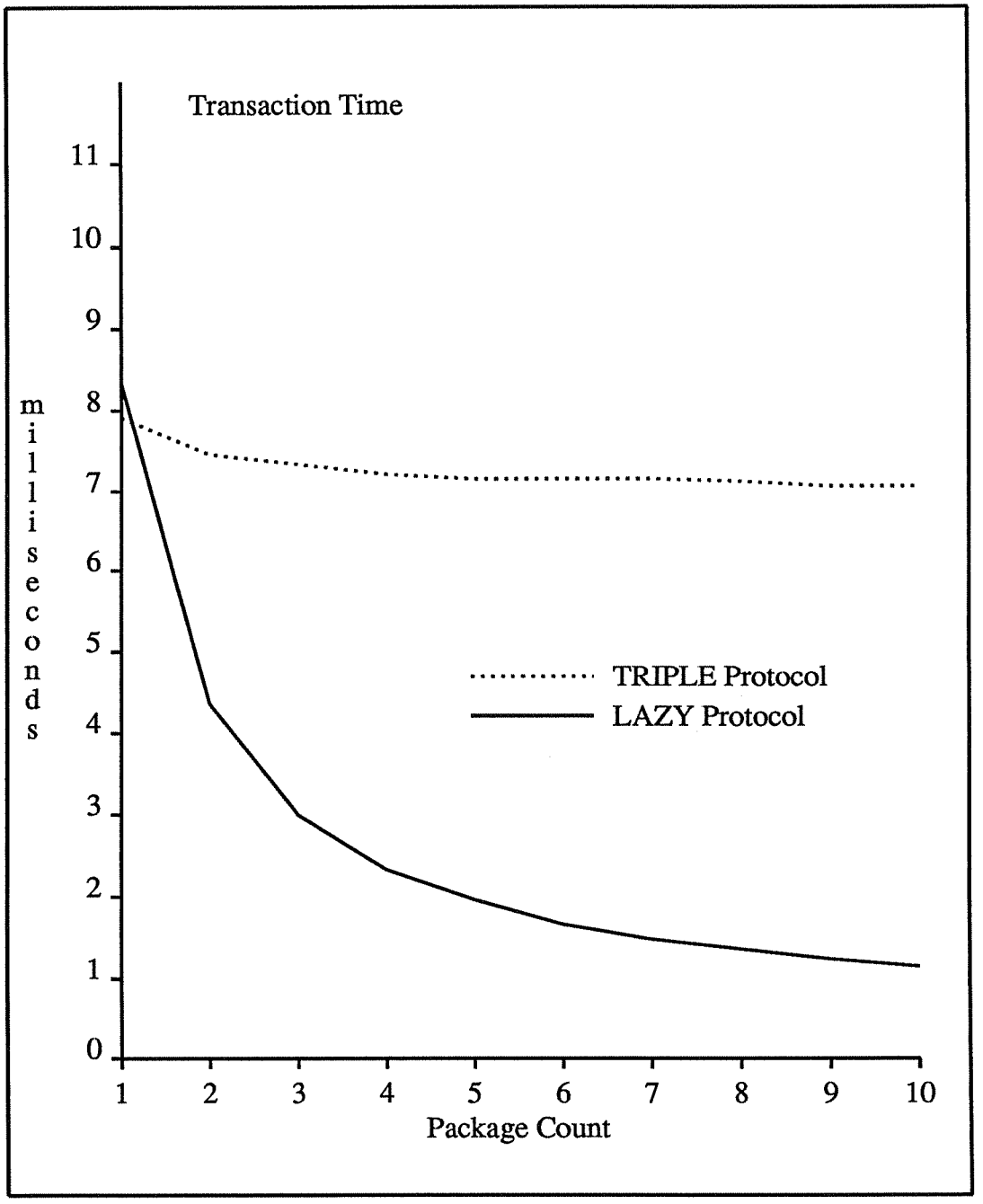

Figure 6.7: Packaged producer-consumer – transaction time vs. package count

average over 2520 transactions. This number is the smallest integer evenly divisible by all the integers between 1 and 10 inclusive.

The TRIPLE protocol version is able to take some advantage of the larger package counts because its overhead per event is reduced, but it always sends the same number of messages so its performance increases only marginally as more events are packaged in each regular expression. The LAZY protocol, on the other hand, can fit all the packaged objects in a single message, so its performance improves dramatically with package count.

Figure 6.8 shows the same example, but this time varying the object size while keeping the package count fixed at 100 objects per package. Object size is varied in 100 byte increments, and each datum is an average over 1000 transactions, which amounts to 10 loop iterations.

As objects get larger, fewer and fewer will fit in a single low-level message. Above 1000 bytes only one object will fit in each message, so in this range the LAZY and TRIPLE protocols perform comparably, the LAZY protocol suffering an overhead of about 0.3 milliseconds per transaction. On smaller objects the LAZY protocol significantly outperforms the TRIPLE protocol. Three of these object sizes allow two objects per message, and two of them allow three objects per message. This observation accounts for the plateaus that occur in the lower portion of the LAZY protocol curve.

6.3.5 Buffered Producer-Consumer Example

Our last example also involves a producer-consumer conversation, but in this case the client processes buffer their requests, that is, they begin working on the next request without first waiting for the previous request to complete. The producer executes the following code fragment:

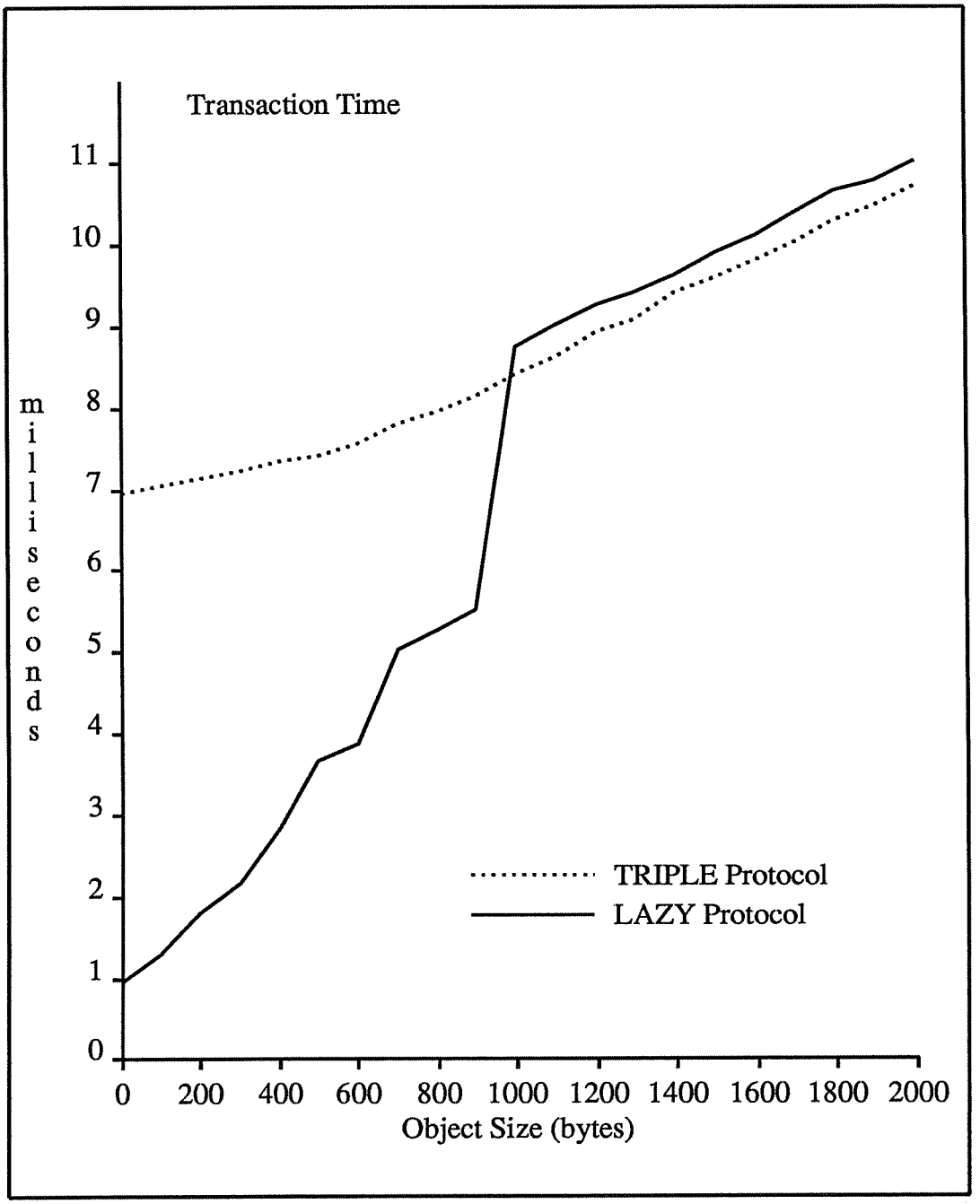

Figure 6.8: Packaged producer-consumer – transaction time vs. object size

```
putexpr := PutItem (conv, Object);
Produce (Object);
Append (putexpr, plan);
for i := 1 to Count - 1 do
    Produce (NextObject);
    Wait ();
    Object := NextObject;
    Append (putexpr, plan);
end;
Wait ();
FreeExpression (putexpr);
```

The consumer executes the corresponding fragment:

```
getexpr := GetItem (conv, Object);
Append (getexpr, plan);
for i := 1 to Count - 1 do
    Wait ();
    LastObject := Object;
    Append (getexpr, plan);
    Consume (LastObject);
end;
Wait ();
Consume (Object);
FreeExpression (getexpr);
```

As before, we can vary the size of objects and the time it takes to produce and consume objects.

Figure 6.9 shows the time per iteration of this example as a function of producer time, assuming four-byte objects and a consumer time of zero. Producer time is varied in 1 millisecond increments, and each datum is an average over 100 loop iterations.

This example illustrates all three of the conversation-based protocol improvement strategies discussed in section 5.2. We will examine the features of this figure in order of increasing producer time.

For very small producer times, the two clients generate events simultaneously,

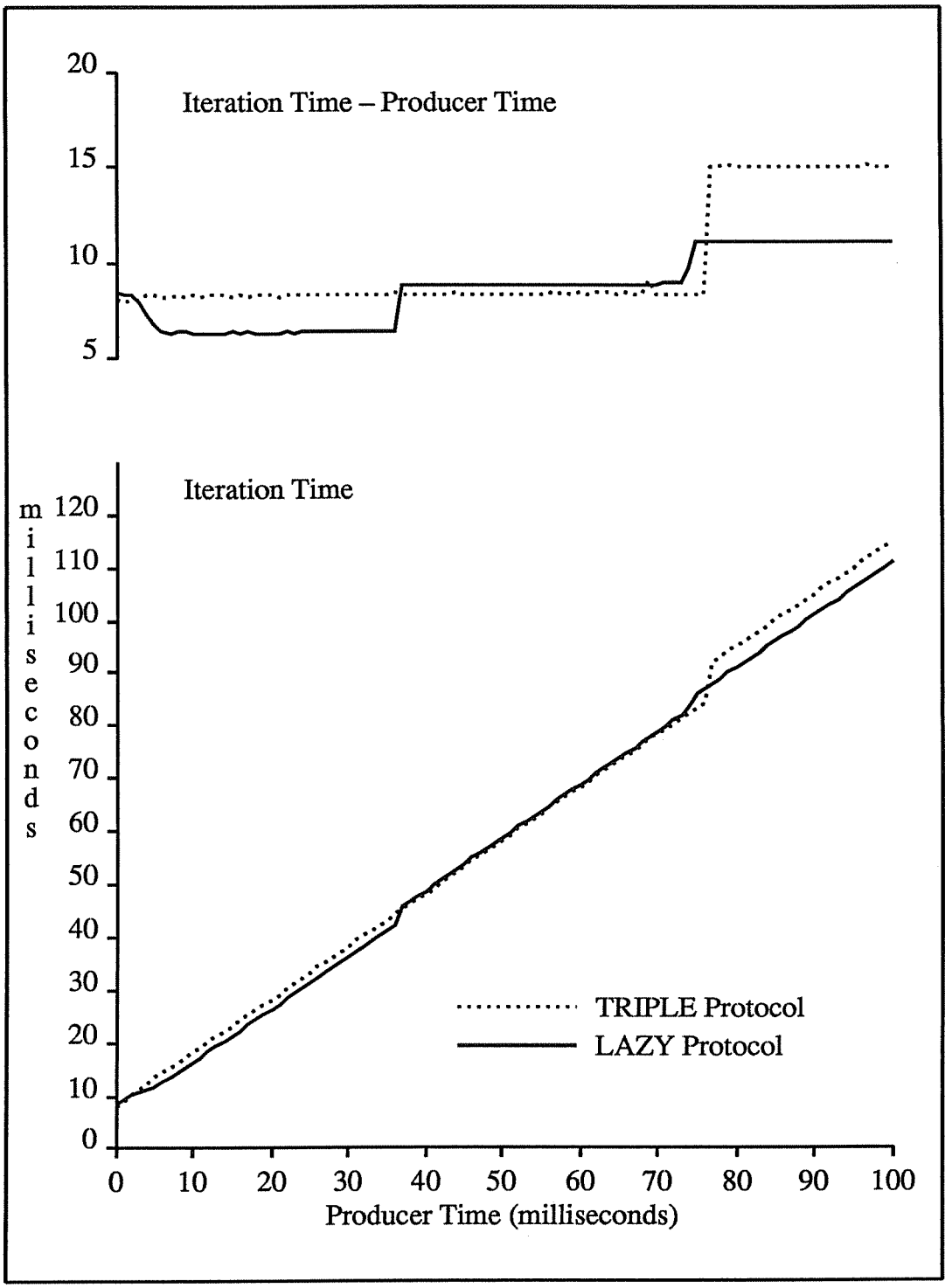

Figure 6.9: Buffered producer-consumer – iteration time vs. producer time

and since both events are specified decisively the conversation tasks send crossing **request** messages. None of the message delay strategies are applicable in this mode, so the TRIPLE protocol outperforms the LAZY protocol by an amount equal to the overhead for collecting statistics and searching for improvement opportunities.

When the producer takes at least 5 milliseconds to produce each object, the consumer's **request** message arrives before the producer generates the matching event. When the event is generated, the producer's conversation task has the option of sending an **accept** message immediately or delaying it and later sending a combined **accept-request** message. The criteria for choosing the latter course are met in this example because the producer is double buffering its objects. The consumer's conversation task can respond to the combined **accept-request** message with an **accept-request** message of its own, because the response will arrive before the producer needs the **accept**. Therefore the LAZY protocol uses one message per iteration in the producer time range of about 5 to 35 milliseconds. Each message completes one transaction and initiates the next. In this range the LAZY protocol uses less than 6.4 milliseconds per iteration, excluding the time it takes to produce objects. This time represents a 23 percent improvement over the TRIPLE protocol.

At a producer time of about 35 milliseconds, the producer's conversation task can no longer delay **accept** messages because the time to produce two objects begins to exceed the 75 millisecond *a priori* bound on how long messages may be delayed. However, it can still delay its response to the consumer's **request** message until one object is produced, so it can send **accept** messages rather than **wait** and **resume** messages. In the producer time range of about 35 to 75 milliseconds the LAZY protocol's performance is slightly less than that of the TRIPLE protocol with its fixed 75 millisecond **wait** delay.

Near a producer time of 75 milliseconds both protocols switch to **wait-resume**

mode, but here, as in the first example, the LAZY protocol is able to include the **ack** messages of that mode in other **request** messages. Above 75 milliseconds the LAZY protocol outperforms the TRIPLE protocol by about 26 percent.

Throughout the range of producer times the performance of the TRIPLE protocol at best slightly exceeds that of the LAZY protocol, and over much of the range the LAZY protocol is significantly better.

6.4 Summary

The performance measurements presented in this chapter can be summarized as follows. In the EXCHANGE protocol implementation using reliable nugget messages, communication overhead is approximately 9.1 milliseconds per transaction plus 0.004 milliseconds per byte of client data transferred. The alternating-bit implementation of the EXCHANGE protocol adds an extra 3.6 milliseconds per transaction. We attribute this additional cost to the fact that the nugget's reliable protocol has more direct access to the communication hardware than the alternating-bit protocol implemented on top of the nugget. The TRIPLE protocol implementation has two major behavior modes, **accept** mode and **wait-resume** mode. In **accept** mode, its communication overhead is approximately 8.4 milliseconds per transaction plus 0.002 microseconds per byte of client data transferred. The overhead is increased by approximately 6.9 milliseconds per transaction in **wait-resume** mode.

The LAZY protocol implementation incurs the additional overhead associated with maintaining runtime statistics and searching for improvement opportunities. The extra cost ranges from 0.4 to 0.9 milliseconds per transaction, which represents an increase of 5 to 10 percent over the communication overhead of the TRIPLE

protocol implementation. This increased overhead is offset by the performance gain realized when one or more of the protocol improvement strategies are successful. The plan-based improvement strategy yields net performance improvements as high as 84 percent, while the conversation-based strategies yield net improvements as high as 26 percent. These results show that application-specific information can often be used to improve the application's communication efficiency. We cannot calculate an overall "average" improvement because we do not have any notion of what constitutes an average application program.

The implementation effort described in this chapter has been valuable, not so much because it allowed us to obtain concrete performance measurements, but because it brought to our attention flaws in the basic design of the protocol system itself and in our protocol improvement algorithms. For example, early versions of the design did not include the function `Wait`. Application processes had to use the procedure `Sleep`, which blocks the calling process until all outstanding regular expressions have been completed and deallocated. A process could not reuse a regular expression, nor could it maintain a buffer of two or three outstanding expressions. We did not discover these design shortcomings until we had built an implementation advanced enough to take advantage of the additional information sophisticated application processes could provide. Problems we encountered during the implementation also led us to realize how easily delaying certain messages can lead to decreased client parallelism. This realization forced us to expand the set of statistics maintained at runtime and to increase the complexity of the message-delay decision algorithms. The performance measurements presented in this chapter are not surprising because all the surprising numbers we encountered during the course of the implementation led to modifications of the system design or to changes in the improvement algorithms.

Chapter 7

Future Work

The research described in this dissertation can be extended in several directions.

7.1 Improved Algorithms

The protocol improvement strategies outlined in chapter 5 make use of some of the information contained in plans and conversation descriptions, but they also miss many improvement opportunities.

The plan-based improvement strategy described in section 5.1 attempts to package nontrivial plan fragments in each low-level message. The algorithm considers only fragments that consist entirely of communication events that all belong to the same conversation. It should be possible to make effective use of fragments that include action procedure calls, although it might be necessary to maintain runtime estimates of the running times of such procedures. Fragments that involve more than one conversation may also be useful. Two communication events belonging to one conversation, but separated by an event belonging to a different conversation, could still be packaged in a single message provided the intervening event is completed quickly. Runtime statistics could be used to recognize such situations.

The conversation-based improvement strategies described in section 5.2 can also be improved. Better runtime characterizations of client behavior would allow more accurate improvement decisions. Also, better algorithms may be able to recognize more improvement opportunities, given the client statistics that are currently maintained.

7.2 Improved Conversation Descriptions

More protocol improvements might be possible if conversation descriptions contained more information.

A major shortcoming of conversation descriptions as they are currently defined is that they contain no timing information. The remote procedure call description says that an `Answer` message will be sent in response to each `Question` message, but it says nothing about how quickly the response will be generated. Therefore the protocol generator cannot always use the `Answer` as an acknowledgement of the `Question` but must instead rely on timing information derived at runtime. Conversation descriptions could include some sort of timing constraints that reflect the expected uses of the conversation. Clients that fail to meet the constraints might receive less efficient service, or they might be aborted.

Conversation descriptions might include other constraints on how a conversation may be used. Knowledge that certain messages in a conversation must always be sent or received decisively could allow the protocol generator to accomplish those messages more efficiently. Such knowledge would allow agents to avoid some of the guesses they currently make.

Finally, conversation descriptions might be extended to include unreliable as well as reliable message classes. Such an extension would make the protocol gen-

erator useful to a wider variety of applications.

7.3 Improved Plans

If plans were more powerful, programmers could make more use of them, giving the system more information on which to base improvements. Plans are the means by which clients specify their intentions to their agents, and the more completely those intentions are specified the more chance the agents would have of improving their clients' communication efficiency. Several operators in addition to the regular expression operators `Sequence`, `Choice`, and `Cycle` might be useful.

One simple and useful operator would be a definite iterator, analogous to a `for`-loop, that would specify that a regular expression is to be executed a particular number of times. As plans are currently defined, a client that wants to send exactly a thousand messages must either build an enormous linear plan, or it must execute a smaller plan in an explicit loop. Making such loop information visible to the agents should allow them to execute plans more efficiently.

Other useful operators might be analogues of `if`-statements and `while`-loops. Such operators would require a new type of primitive plan element that might be called a "condition". A condition would be a Boolean function similar to an action procedure that would be called during the execution of a plan to determine which of the alternate expressions in an `if`-statement to execute or whether to exit a `while`-loop. These operators would allow programmers to describe their intentions very precisely. Runtime statistics could be used to estimate the likely results of condition functions. Agents could base their actions on the estimates, but they must always be able to recover cleanly from bad guesses. This problem is similar to the difficulty faced by hardware architects in designing machines that prefetch

instructions or data, and that therefore need to predict the outcome of conditional branch instructions.

These enhancements give plans much of the power of programming languages. The next logical step would be to provide a syntactic interface to plans, rather than the cumbersome procedural interface currently provided. Standard infix regular expression operators could be used, and program fragments could be included directly in plans, eliminating the necessity of action procedures. The compiler for this language could then derive useful information about loops and conditional statements in application programs.

7.4 Relaxed Semantics

The current semantics of conversations and plans, which are derived from CSP, make each high-level message a synchronization point for the processes involved. This strictness precludes some improvements that would otherwise be possible. Many application programs do not depend on the strict synchronization provided by the semantics. For example, the producer process in a producer-consumer example would not ordinarily mind if the system buffers several of the produced objects, so long as they eventually reach the consumer. We might alter the semantics of plans to allow decisively specified `Put`⟨*class*⟩ communication events to complete before the matching `Get`⟨*class*⟩ events have been submitted. This change would allow a greater degree of parallelism between the clients at opposite ends of a conversation. It would allow some of the conversation-based protocol improvement strategies to succeed in more situations, including situations that currently require clients to double buffer their requests. Clients that require the synchronization provided by the old semantics could use `Trade`⟨*class*⟩ events that are synchroniz-

ing by nature. Unfortunately, relaxing the semantics in this manner breaks the symmetry of sends and receives that is one of the appealing aspects of CSP.

Another powerful feature of the semantics of our system and of CSP is the "exactly one" property of alternative clauses. This property promises that one and only one of the possible alternative communication events at any point in a plan will be accomplished. Once again, many clients could as easily use an "at least one" property that could be implemented more efficiently. For example, a bounded buffer process is usually willing to complete either a `GetItem` or `PutItem` communication event, but it can just as easily tolerate the simultaneous completion of both events. However, clients that rely on the current semantics would be considerably more complicated if less powerful semantics were provided. A good example is the stream-oriented file server of Figure 3.6. The heart of the server's plan is a choice between `PutData`, `GetData`, and `GetSeek` communication events. The server cannot tolerate the simultaneous completion of both the `PutData` and `GetSeek` events, because the action following the reception of a seek message modifies the outgoing data buffer. The client may get the wrong data. Without the "exactly one" property of alternative clauses, file servers and clients would have to follow a more complicated handshaking procedure to correctly handle seek requests. Perhaps an "at least one" variant of the `Choice` operator could be provided for those clients that do not require the power of the current semantics.

7.5 Better Theoretical Foundation

The *ad hoc* algorithms of section 5.2 for choosing an appropriate **wait**, **ack**, or **accept** delay should be placed on a better mathematical foundation. These algorithms attempt to recognize situations in which delaying a particular message

would be profitable, but they involve arbitrary constants such as the parameter k that controls the number of measurements included in statistics, and the *a priori* upper bound on how long any message may be delayed. Furthermore, they use the minimum and maximum of the last k values of a variable to characterize the variable, when other functions of the past values might be more appropriate.

We can at least make a step toward putting these algorithms on a better theoretical basis. For example, consider the choice of an appropriate **wait** delay in the LAZY protocol. A conversation server that has received a **request** message and whose plan driver does not currently have a matching communication event must decide whether, and by how much, to delay its **wait** message in hopes of sending an **accept** message instead. If we could assign an expected cost to each possible **wait** delay, we could choose the delay that minimizes the expected cost.

The expected cost has two components. First, if we choose a delay that is too short, we incur the costs of sending two additional messages. Second, the longer the delay we choose, the longer it takes to recover from lost messages, because the original server must allow time for the **wait** delay and a message roundtrip before retransmitting its **request** message.

These statements must be quantified to be useful. Let $C(d)$ be the cost function that assigns an expected cost to each possible **wait** delay d, $d \geq 0$.

The first component of this cost is $2M$ if d is too small and 0 otherwise, where M is the message transmission time of the underlying communication system. Let W measure the length of time between the arrival of the **request** message and the client's submission of a matching communication event. Then the first component of the expected cost is $2M \Pr[W > d]$, where $\Pr[W > d]$ is the probability that W exceeds d. We can regard W as a nonnegative random variable with distribution function F_W, in which case the expected cost's first component can be written as

$2M(1 - F_W(d))$.

To a first approximation, the second component of the cost is 0 if no messages are lost and $2M + d$ otherwise, because the original server's retransmission interval must exceed $2M + d$. Let L be the probability that the underlying communication system loses any particular message. Then the second component of the expected cost is simply $L(2M + d)$, ignoring the possibility of two or more lost messages.

We can therefore write the expected cost function $C(d)$ as:

$$C(d) = 2M(1 - F_W(d)) + L(2M + d)$$

or

$$C(d) = 2M(1 + L) - 2MF_W(d) + Ld$$

In this formula, M and L are constant characteristics of the underlying communication system, and F_W is a characteristic of the client's behavior in this situation.

Figure 7.1 shows the expected cost function for three different values of the message-loss probability L, assuming a message transmission time M of 2 milliseconds and that W is normally distributed with a mean of 50 milliseconds and a standard deviation of 5 milliseconds.

Once the expected cost function has been defined, the location of its minimum value can be determined. The general shape of the expected cost curve is determined by the shape of the probability distribution function F_W. All distribution functions of nonnegative random variables start somewhere between 0 and 1 at the origin and increase monotonically to 1 (perhaps asymptotically). Therefore the expected cost function includes a constant term $2M(1 + L)$, a monotonically decreasing term $-2MF_W(d)$ and a monotonically increasing term Ld. The location of the minimum expected cost is not affected by the constant term, but is determined by the interplay of the decreasing and increasing terms. If the function is smooth,

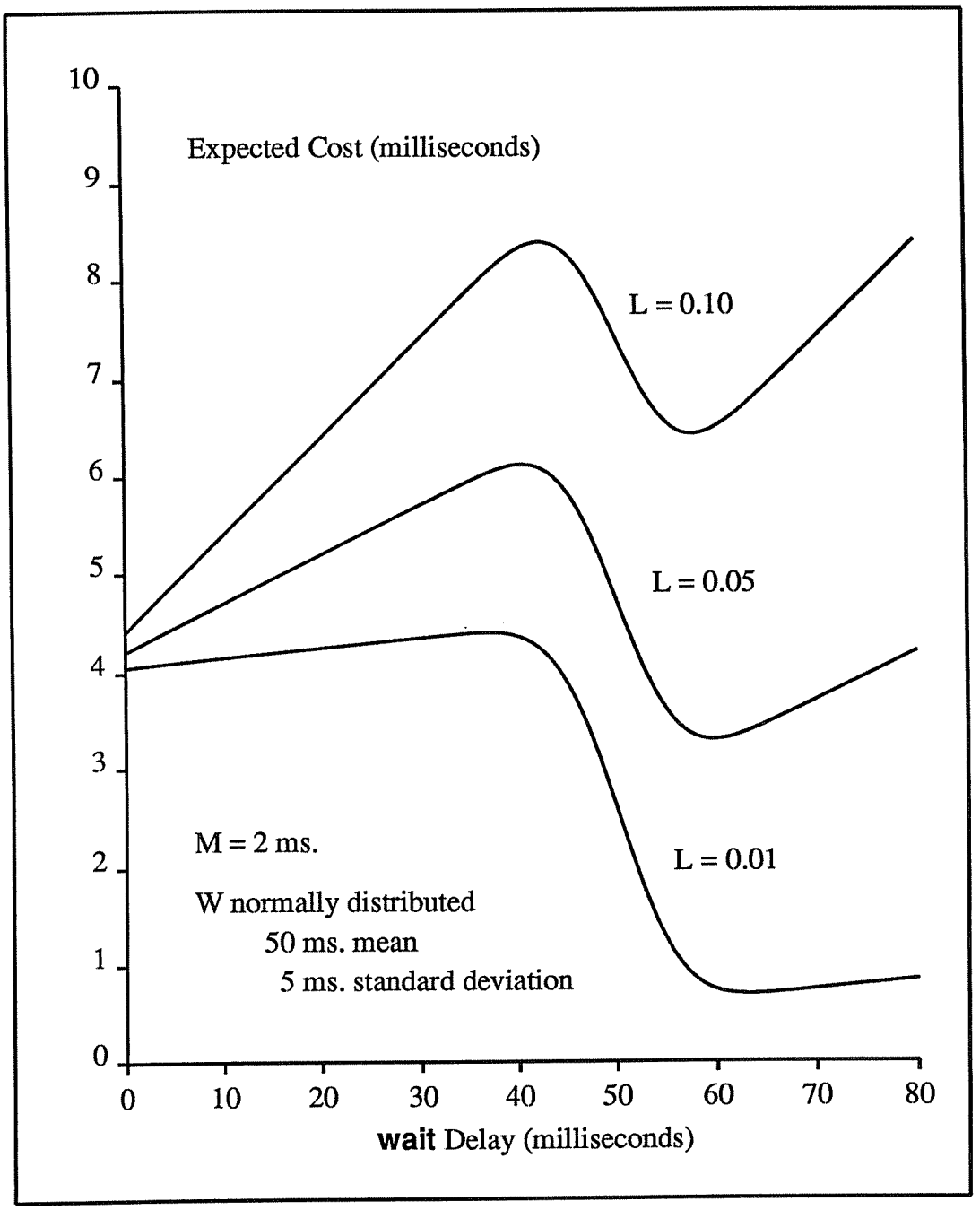

Figure 7.1: The expected cost of delaying a **wait** message

its minimum value must occur at the origin or at a point where the function's first derivative is 0. Otherwise the minimum value may also occur at a point where the function's first derivative is undefined.

Cost functions that involve a standard continuous distribution, such as the normal, the exponential, or the uniform distribution, have at most one local minimum other than the origin. The decision as to whether to delay a **wait** message then boils down to the question of whether the expected cost at that local minimum is less than the expected cost at the origin. If not, the **wait** should be sent immediately. Otherwise it should be delayed by an amount determined by the location of the local minimum.

For example, if W is normally distributed with mean μ and variance σ^2, then

$$F_W(d) = \int_{-\infty}^{d} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2}(x-\mu)^2/\sigma^2} dx$$

and $C(d)$ has a local minimum at the delay

$$d_{\min} = \mu + \sqrt{-2\sigma^2 \ln\left(\frac{L\sqrt{2\pi\sigma^2}}{2M}\right)}$$

provided $L\sqrt{2\pi\sigma^2} < 2M$. The **wait** message should be delayed if $C(d_{\min}) < C(0)$ and sent immediately otherwise. Unfortunately, this criterion cannot be evaluated cheaply since it involves several transcendental functions and a definite integral that does not have a closed functional form.

We have now reduced the problem of choosing an appropriate **wait** delay to the problem of determining the distribution function F_W of the random variable W that measures the length of time between the arrival of the **request** message and the appearance of a matching communication event. An estimate of the distribution function can be based on the measurements of the client's past behavior. A

large body of statistical machinery has been developed for exactly the purpose of estimating the distribution of a random variable from a set of observations of the variable's value [Mood74]. The standard approach would be to assume some form for the variable's distribution and to use past values of the variable to estimate the parameters that characterize a particular member of the assumed family of distributions. For example, we might assume that W is normally distributed, in which case the sample mean and sample variance are our best estimators of the parameters μ and σ^2 that characterize the normal distribution. (It is interesting to note that the best estimators of the parameters of the uniform distribution are the sample minimum and the sample maximum. Perhaps an implicit assumption of uniformity underlies the decision strategies described in section 5.2.) We must still allow for the fact that W's distribution may vary with time, so parameter estimates based on the last k observations or on a geometrically weighted average of the past observations might be appropriate.

This analysis places the choice of an appropriate **wait** delay on a firm theoretical foundation. We simply choose the delay that minimizes our expected costs. Unfortunately, the cost function described in the last few paragraphs is only an approximation to the true cost function. A more accurate function would include the costs of handling timers, it would better reflect the costs incurred when messages are lost, and it would allow for interactions with other improvement attempts. For example, a lost **request** message costs nothing if it can be retransmitted and delivered before the matching communication event is available, and the cost of choosing an insufficient **wait** delay is only one extra message if the final **ack** can be included in a later message.

Ideally, we would analyze an entire conversation and choose parameter values that jointly minimize the conversation's expected execution time. We would deco-

rate the conversation's DFA with estimated transition probabilities and estimated distributions of the time intervals between transitions, and from that information we would determine parameter values that are jointly optimal. A step toward this ideal would be to analyze acyclic paths through the DFA in an effort to jointly optimize the parameters involved in the path.

Placing our various protocol improvement strategies on a firmer theoretical and mathematical foundation would allow us to consider "protocol optimization", not just "protocol improvement".

Chapter 8

Conclusions

The research described in this dissertation makes a step in the direction of improving application-level communication efficiency by using information provided by, and derived from, the application itself. It also addresses a gap in the existing hierarchy of protocol development systems.

A communication protocol can be considered at three levels of increasing abstraction, the implementation level, the protocol specification level, and the service specification level. Protocol verification systems attempt to demonstrate that a proposed protocol specification meets some required service specification. Protocol implementation systems construct concrete implementations based on abstract protocol specifications. Our protocol generation system creates a protocol implementation directly from a limited form of service specification we call a conversation description.

A conversation description is a high-level specification of the service provided by an abstract communication channel. It does not include such implementation details as sequence numbers, acknowledgements, or retransmissions. The protocol generator described in this dissertation builds a communication protocol that

provides the service specified by a conversation description. A deterministic finite automaton (DFA) drives each end of the constructed protocol. The automaton is the product of two DFA's, one constructed from the conversation description's regular expression and another that incorporates sequence numbers, retransmission timers, and acknowledgements. The latter DFA is a common component of all constructed protocols. The conversation-specific component enriches the state space of the constructed protocol, making the protocol more adaptable to its environment than standard protocols. The enriched state space lets us maintain a detailed characterization of the runtime behavior of the application process. Using this characterization, we can predict the success or failure of protocol improvement strategies, and we can select appropriate values for protocol parameters.

We have defined an interprocess communication mechanism we call a plan. Application processes use plans dynamically to participate in conversations. Plans are a generalization of CSP's communication mechanism. They allow all regular expression operators, not just alternation, and they relax CSP's prohibition of output statements in guards. We allow indecisively specified output events, which are output events that appear at choice points in plans, but we define the event-matching criteria in such a way that each pair of matching communication events must include at least one decisively specified event. This relaxed restriction restores the symmetry of input and output events and also allows an efficient implementation. The TRIPLE protocol presented in chapter 4 is an efficient low-level communication protocol we use to implement plans. It could also be used to implement a version of CSP that allows both input and output guards but that requires at least one of each matching pair of communication statements to be deterministic.

Plans are another source of application-specific knowledge that can be used to increase the efficiency of the application's communication. They contain infor-

mation about the immediate and future communication intentions of application processes. Knowledge of future intentions often allows us to use fewer low-level messages to accomplish the intended communication than would be necessary if the knowledge were not available.

Our implementation of the protocol generator supports application processes running on the Crystal multicomputer. We have evaluated the performance of several typical applications. The overhead associated with making runtime measurements and searching for improvement opportunities ranges from 5 to 10 percent of the time needed to accomplish a single communication event involving no data. The various improvement strategies yield performance gains ranging from 0 to 84 percent.

We believe these results justify our contention that application-level communication efficiency can be significantly improved using information provided by, and derived from, the application itself.

References

[Aho77] **Aho, A. V. and J. D. Ullman,** *Principles of Compiler Design*, Addison-Wesley, 1977.

[Anderson85] **Anderson, D. P.,** Protocol Specification by Real-Time Attribute Grammars, Ph.D. Thesis, 1985

[Bernstein80] **Bernstein, A. J.,** "Output Guards and Nondeterminism in 'Communicating Sequential Processes'," *ACM Transactions on Programming Languages and Systems* 2:2 (April 1980), pp. 234-238.

[Bochmann80] **Bochmann, G. V.,** "A General Transition Model for Protocols and Communication Services," *IEEE Transactions on Communications* 28:4 (April 1980), pp. 643-650.

[Brand78] **Brand, D. and W. H. Joyner, Jr.,** "Verification of Protocols using Symbolic Execution," pp. 351-360 in *Computer Networks*, North-Holland, 1978.

[Buckley83] **Buckley, G. N. and A. Silberschatz,** "An Effective Implementation for the Generalized Input-Output Construct of CSP," *ACM Transactions on Programming Languages and Systems* 5:2 (April 1983), pp. 223-235.

[Cheriton83] **Cheriton, D. R. and W. Zwaenepoel,** "The Distributed V Kernel and its Performance for Diskless Workstations," *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, 10-13 October 1983, pp. 128-139. In *ACM Operating Systems Review* 17:5.

[Chow85] **Chow, C. H., M. G. Gouda, and S. S. Lam,** "A Discipline for Constructing Multiphase Communication Protocols," *ACM Transactions on Computer Systems* 3:4 (November 1985), pp. 315-343.

[Clarke86] **Clarke, E. M., E. A. Emerson, and A. P. Sistla,** "Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications," *ACM Transactions on Programming Languages and Systems* 8:2 (April 1986), pp. 244-263.

[Cook83] **Cook, R., R. Finkel, D. DeWitt, L. Landweber, and T. Virgilio**, "The Crystal Nugget: Part I of the First Report on the Crystal Project," Technical Report 499, Computer Sciences Department, University of Wisconsin - Madison, April 1983.

[Danthine80] **Danthine, A. A. S.**, "Protocol Representation with Finite-State Models," *IEEE Transactions on Communications* 28:4 (April 1980), pp. 632-643.

[Finkel83] **Finkel, R., M. Solomon, D. DeWitt, and L. Landweber**, "The Charlotte Distributed Operating System: Part IV of the First Report on the Crystal Project," Technical Report 502, Computer Sciences Department, University of Wisconsin - Madison, July 1983.

[Good77] **Good, D. I.**, "Constructing Verified and Reliable Communications Processing Systems," *ACM SIGSOFT Software Engineering Notes* 2:5 (October 1977), pp. 8-13.

[Hoare78] **Hoare, C. A. R.**, "Communicating Sequential Processes," *CACM* 21:8 (August 1978), pp. 666-677.

[Ichbiah79] **Ichbiah, J. D. et al.**, "Preliminary Ada Reference Manual," *Sigplan Notices* 14:6 (June 1979).

[Lampson80] **Lampson, B. W. and D. D. Redell**, "Experience with Processes and Monitors in Mesa," *CACM* 23 (February 1980), pp. 105-117.

[Leblanc82] **Leblanc, T. J.**, The Design and Performance of High-Level Language Primitives for Distributed Programming, Ph.D. thesis, 1982

[Linn83] **Linn, R. J., J. S. Nightingale, and W. H. McCoy**, "Testing OSI Protocols: A Compendium of Papers," Report No. ICST/SNA - 83-1, National Bureau of Standards, Washington, D.C., June 1983.

[Merlin76] **Merlin, P. M.**, "A Methodology for the Design and Implementation of Communication Protocols," *IEEE Transactions on Communications* 24:6 (June 1976), pp. 614-621.

[Molloy82] **Molloy, M. K.**, "Performance Analysis Using Stochastic Petri Nets," *IEEE Transactions on Computers* 31:9 (September 1982), pp. 913-917.

[Mood74] **Mood, A. M., F. A. Graybill, and D. C. Boes**, *Introduction to the Theory of Statistics*, McGraw-Hill, 1974.

[Nash83] **Nash, S. C.**, "Automated Implementation of SNA Communication Protocols," *IEEE International Conference on Communication*, June 1983.

[Postel81] **Postel, J.**, "Transmission Control Protocol - DARPA Internet Program Protocol Specification," RFC 793, Information Sciences Institute, University of Southern California, September, 1981.

[Ramamoorthy85] **Ramamoorthy, C. V., S. T. Dong, and Y. Usuda**, "An Implementation of an Automated Protocol Synthesizer (APS) and Its Application to the X.21 Protocol," *IEEE Transactions on SoftWare Engineering* SE-11:9 (September 1985), pp. 886-908.

[Rand-Corporation80] **Rand Corporation**, "Formal Methods for Communication Protocol Specification and Verification," Report No. ICST/HLNP - 80-7, National Bureau of Standards, Washington, D.C., June 1980. Draft Report.

[Saltzer84] **Saltzer, J. H., D. P. Reed, and D. D. Clark**, "End-To-End Arguments in System Design," *ACM Transactions on Computer Systems* 2:4 (November 1984), pp. 277-288.

[Sidhu83] **Sidhu, D. P. and T. P. Blumer**, "An Automated Protocol Development System," *IEEE Symposium on Application and Assessment of Automated Tools for Software Development*, 1983, pp. 10-23.

[Spector82] **Spector, A. Z.**, "Performing Remote Operations Efficiently on a Local Computer Network," *CACM* 25:4 (April 1982), pp. 246-260.

[Sunshine81] **Sunshine, C. A.**, "Formal Modeling of Communication Protocols," RR-81-89, Information Sciences Institute - University of Southern California, March 1981.

[Tanenbaum81] **Tanenbaum, A.**, *Computer Networks*, Prentice-Hall, 1981.

[Teng78] **Teng, A. Y. and M. T. Liu**, "A Formal Model for Automatic Implementation and Logical Validation of Network Communication Protocol," *NBS Computer Networking Symposium, IEEE*, 1978, pp. 114-123.

[Thompson81] **Thompson, D. H., C. A. Sunshine, R. W. Erickson, S. L. Gerhart, and D. Schwabe**, "Specification and Verification of Communication Protocols in AFFIRM Using State Transition Models," RR-81-88, Information Sciences Institute - University of Southern California, March 1981.

[West78] **West, C. H.**, "General Technique for Communications Protocol Validation," *IBM Journal of Research and Development* 22:4 (July 1978), pp. 393-404.

[Wirth82] **Wirth, N.**, *Programming in MODULA-2*, Springer-Verlag, 1982.

[Zafiropulo78] **Zafiropulo, P.**, "Protocol Validation by Duologue-Matrix Analysis," *IEEE Transactions on Communications* 26:8 (August 1978), pp. 1187-1194.