Flush communication channels: Effective implementation and verification

Tracy Kay Camp
College of William & Mary - Arts & Sciences

Flush communication channels: Effective implementation and verification

Camp, Tracy Kay, Ph.D.

The College of William and Mary, 1993

FLUSH COMMUNICATION CHANNELS: EFFECTIVE IMPLEMENTATION AND VERIFICATION

A Dissertation

Presented to

The Faculty of the Department of Computer Science

The College of William and Mary in Virginia

In Partial Fulfillment

Of the Requirements for the Degree of

Doctor of Philosophy

by

Tracy Camp

1993

APPROVAL SHEET

This dissertation is submitted in partial fulfillment of

the requirements for the degree of

Doctor of Philosophy

Tracy Camp

Tracy Camp

Approved, August 1993

[illegible]

Phil Kearns, Thesis Advisor

Stephen Park

Stephen Park

[illegible]

William Bynum

[illegible]

Robert Noonan

[illegible]

Stephen Knudson

I dedicate this dissertation to my parents, Sam and Betty Camp.

Contents

ACKNOWLEDGEMENTS

I am extremely indebted to my advisor Phil Kearns, "P", for his guidance, criticism, and patience in the creation of this document. He has been instrumental in molding this student into an effective researcher. Furthermore, my thanks go to the committee members for their time on my behalf. I am also grateful to the Patricia Roberts-Harris Fellowship and the National Science Foundation Grant CCR-8705603 for financial support.

For supporting me throughout my years at William and Mary, many thanks go to several friends. Mike Seale for his diligence not only in problem solving, but in attempting to win on the racquetball court; Marge Cook for the low rent and late-night discussions; Tracey Beauchat for all his system expertise; Steph Lake for bar hopping and sharing dreams; Matt Johnson for introducing me to this university and keeping me amused during my tenure here; Laurie King for a special friendship along with crisis intervention and prevention; and Glen Oberhauser for pushing me to stay on top of my dissertation and keeping me stable during the stressful completion of this document.

In addition, thanks to my Uncle Dave who was my role model in becoming part of the academic world. And lastly, I'll forever be indebted to my parents.

List of Figures

ABSTRACT

Flush communication channels, or F-channels, generalize more conventional asynchronous communication paradigms. A distributed system which uses an F-channel allows a programmer to define the delivery order of each message in relation to other messages transmitted on the channel. Unreliable datagrams and FIFO (first-in-first-out) communication channels have strictly defined delivery semantics. No restrictions are allowed on message delivery order with unreliable datagrams—message delivery is completely unordered. FIFO channels, on the other hand, insist messages are delivered in the order of their transmission. Flush channels can provide either of these delivery order semantics; in addition, F-channels allow the user to define the delivery of a message to be after the delivery of all messages previously transmitted or before the delivery of all messages subsequently transmitted or both. A system which communicates with a flush channel has a message delivery order that is a partial order.

Dynamically specifying a partial message delivery order complicates many aspects of how we implement and reason about the communication channel. From the system's perspective, we develop a feasible implementation protocol and prove its correctness. The protocol effectively handles the partially ordered message delivery. From the user's perspective, we derive an axiomatic verification methodology for flush applications. The added flexibility of defining the delivery order dynamically slightly increases the complexity for the application programmer. Our verification work helps the user effectively deal with the partially ordered message delivery in flush communication.

FLUSH COMMUNICATION CHANNELS:
EFFECTIVE IMPLEMENTATION AND VERIFICATION

Good communication is as stimulating as black coffee,
and just as hard to sleep after.
Anne Morrow Lindbergh

Chapter 1

Introduction

1.1 Communication Paradigms

A distributed system is a set of processes which communicate via message passing. Communication is termed asynchronous when a message **send** operation does not wait for execution of its matching **receive**. The network subsystem which implements the communication path between the sender and the receiver handles the message until delivery occurs at the destination process [Tan89].

One way to categorize asynchronous message passing constructs is by the delivery order restrictions placed upon messages. For example, unreliable datagram communication imposes no delivery order restriction. The fact that datagram m_1 is transmitted before m_2 says nothing about the order in which the destination process may receive those messages (if they are received at all). A virtual circuit or sequenced reliable packet protocol, on the other hand, imposes a rigid order on message delivery. If message m_1 is transmitted before message m_2 over a FIFO (first-in-first-out) channel, then m_1 must be delivered before m_2. There is, however, no reason to impose all-or-nothing delivery order requirements on all messages in every distributed program.

Flush channels generalize the above two competing views of asynchronous message passing semantics by allowing a programmer to specify message delivery order restrictions as appropriate to the needs of the program [Ahu90]. An F-channel is unidirectional and reliable. The **send** construct appears as:

send (type, data) **on** F

where type is the F-channel message type, data is the data to be transmitted, and F is the identity of the F-channel connecting two communicating processes. Four message types, each with a different impact upon delivery order, are available to the programmer using F-channel communication:

- A *two-way flush* message (type **2F**) flushes the communication channel in two directions. The two-way flush message is delivered after every message transmitted before it and before every message transmitted after it.
- A *forward flush* message (type **FF**) flushes the channel in a forward direction. The message is delivered after every message transmitted before it.
- A *backward flush* message (type **BF**) flushes the channel in a backward direction. The backward flush message is delivered before every message transmitted after it.
- An *ordinary* message (type **Ord**) does not flush the communication channel at all. The only constraints on the delivery of an ordinary message are those imposed by the other three message types.

It is important to note that the ordering restrictions placed upon messages concerns their *delivery* to the destination user process; flush messages may *arrive* at the destination host in any order. We use the term *arrival* to denote the point in time that a message is passed from the network to the destination host. The term *delivery* represents the event when the message becomes eligible for reception by the destination process. A correct implementation of an F-channel ensures messages arriving at the destination process are delivered without violating the order specified by the transmitting process.

The **receive** operation for F-channel communication appears as:

receive (type, data) **from** F.

The invoking destination process of the **receive** operation is blocked until there is a message available for delivery on F.

Programming distributed systems which use F-channels is relatively straightforward. As illustrated in [Ahu90], such programming may be as simple as that which relies upon virtual circuit communication. A global snapshot protocol, similar in spirit to that of Chandy and Lamport [CL85] but cast into the context of a system which uses F-channel communication, is presented in [Ahu90]. Communication paradigms for F-channels are developed in [AVS91]. In all of these examples, the use of F-channels provides a greater potential for concurrency than the use of FIFO channel communication but without the programming disadvantages of unreliable datagram communication. This increase in concurrency of message delivery can occur when there are multiple physical paths between source and destination and packet switched routing is used.

1.2 Formalization of the Flush Channel Delivery Order

To obtain the greater potential for concurrency offered with F-channel communication, we must solve the problem of dealing with the dynamic delivery order. FIFO communication channels and unreliable datagram communication have static delivery order semantics *independent* of the application program which utilizes the channel. The delivery order for an F-channel, on the other hand, is *defined* by the application program. As the program executes, the delivery order is created "on the fly". We need an abstraction to easily understand this dynamic, and possibly complicated, delivery order that is *inherent* in F-channel communication.

Possible delivery orders are interpreted by Ahuja from the perspective of *message crossing* [Ahu90]. Two messages sent on an F-channel are said to cross when they are delivered in an order different from their transmission order. Crossing is allowed when no ordering restrictions exist between two messages. As defined, ordinary messages place no restrictions on the delivery order. A two-way flush, on the other hand, places strict delivery ordering requirements on the channel; no message (of any type) is allowed to cross a two-way flush message. Figure 1.1 illustrates how a two-way flush message, m_{2F}, restricts all crossings. The messages drawn as dashed lines are not allowed. (Real time increases from left to right on the time-lines.)

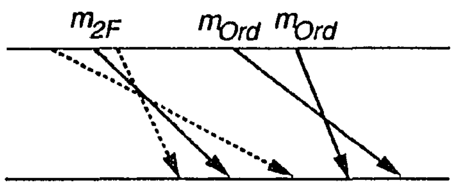

Figure 1.1: Message Crossing: Two-way Flush and Ordinary Messages

A forward flush message, m_{FF}, guarantees all messages sent before m_{FF} are delivered before m_{FF} is delivered. As shown in Figure 1.2, messages sent after m_{FF} are permitted to cross m_{FF} (from right to left). A backward flush message, m_{BF}, guarantees all messages sent after the transmission of m_{BF} are delivered after m_{BF}. Messages sent before m_{BF} are allowed to cross m_{BF} (from left to right).

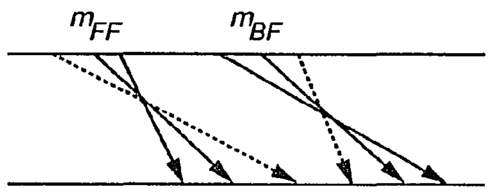

Figure 1.2: Message Crossing: Forward and Backward Flush Messages

The problem with viewing the delivery order of messages in this manner is that the time-lines cannot show the *possibility* of message crossing. The time-lines, instead, show how the messages are *actually* delivered to the destination. For example, consider the two ordinary messages denoted m_{Ord} in Figure 1.1. Since there are no ordering restrictions on these two messages, the first ordinary message may be delivered before or after the second. The time-line, however, can only illustrate one of the two possible delivery order scenarios.

Let M denote the multiset of messages transmitted on F-channel F. We define an irreflexive partial order, $\lhd\!+_F$, on M to represent the inherent delivery order in the system:

$$\lhd\!+_F \subseteq \mathrm{M} \times \mathrm{M},$$

such that for $m, m' \in \mathrm{M}$, $m \lhd\!+_F m'$ if and only if m cannot be delivered after m'. For example, if a two-way flush, m_{2F}, is transmitted before an ordinary message, m_{Ord}, then $m_{2F} \lhd\!+_F m_{Ord}$. We say that m_{2F} is a predecessor of m_{Ord} or, equivalently, m_{Ord} is a successor of m_{2F}. In either case, m_{2F} cannot be delivered after m_{Ord}. A given message may have many predecessors, but if $m, m' \in \mathrm{M}$ and there is no $m'' \in \mathrm{M}$ with $m \lhd\!+_F m'' \lhd\!+_F m'$, then we say that m is an *immediate* predecessor of m'. We define an irreflexive partial order, $\lhd_F$, to represent this immediate predecessor relation. That is, if m is the immediate predecessor of m', then $m \lhd_F m'$. It is possible for elements of M to be unordered under $\lhd\!+_F$. If ordinary message m_{Ord} is transmitted immediately before backward flush m_{BF}, the two messages may be delivered in any order. That is, $m_{Ord} \not\lhd\!+_F m_{BF}$, and $m_{BF} \not\lhd\!+_F m_{Ord}$.

We can draw the immediate predecessor relation, $\lhd_F$, as a directed graph. Each element of M is denoted by a vertex. We connect message m to message m' with a directed edge from m to m' if and only if m is an immediate predecessor of m'. Clearly, $\lhd\!+_F$ is the transitive and irreflexive closure of $\lhd_F$. The graph of immediate predecessors in a partially ordered set conveys all the information about the partial order in a simple manner, including the *possibility* of one message being delivered before another.

Figure 1.3 illustrates an instance of the immediate predecessor abstraction. In this graph, a message is labeled as $<$*type*, $i>$; *type* is the type of the flush message and i is a unique sequence number for the message. We use the notation $<c_1, \ldots, c_k>$ to mean a composite data structure consisting of the elements c_1 through c_k. One may validate the

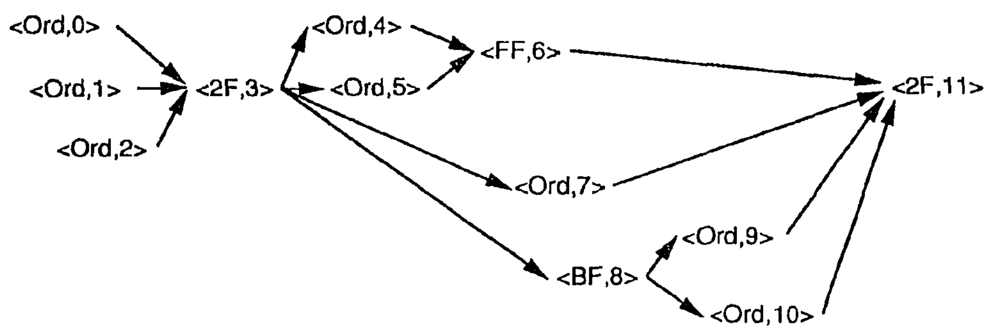

Figure 1.3: A Sample Immediate Predecessor Graph

directed edges from the definition of each flush message type in Section 1.1. For instance, messages numbered 0, 1, and 2 have directed edges to <2F,3>, and there is a path from <2F,3> to all messages with higher sequence numbers. This is in keeping with the definition of a two-way flush message—it must be delivered after every message transmitted before it and before every message transmitted after it.

We find it convenient to define *Pred*(m), the ***predecessor set*** of message m, as

$$Pred(m) = \{m' \in M : m' \lhd\!+_F\, m\}.$$

For example, the backward flush labeled <BF,8> has predecessor set

$$Pred(\text{<BF,8>}) = \{\text{<2F,3>}, \text{<Ord,2>}, \text{<Ord,1>}, \text{<Ord,0>}\}.$$

Every message in *Pred*(<BF,8>) must be delivered before <BF,8>. Messages not in *Pred*(<BF,8>) are unrelated to the delivery of <BF,8>. For example, <Ord,4> may or may not be delivered before <BF,8>. No protocol which implements an F-channel should delay the delivery of <BF,8> due to the non-delivery of <Ord,4>.

1.3 The Problem

Allowing the user to specify the message delivery order complicates implementation and verification. Delivery order semantics are no longer static. The restrictions placed on message delivery are specified by the sender on a message by message basis. The underlying abstraction for the delivery order inherent in F-channel communication, $\lhd_{+F}$, allows us to represent the delivery restrictions simply. We use this irreflexive partial order throughout our consideration of flush communication.

From the system's perspective, an effective implementation that supports a delivery order specified during execution is not obvious. Two attempts at implementing F-channels have appeared in the literature; neither protocol can be considered simple and efficient. We develop an implementation technique that effectively handles the dynamic delivery order. This technique deduces $\lhd_F$ and exploits the deduction to decide when a received message is eligible for delivery. Practical issues, such as finite buffer capacity, are considered as well.

Simulation results illustrate that an F-channel implementation offers the promise of ultra-high bandwidth communication over multiple physical paths. The results obtained are from the special case involving two of the four message types: ordinary messages "batched" with one flush message type. In addition to the simulation results, we provide analysis for the performance of batched ordinary messages as well. The two results validate one another. All implementation results, protocol and performance, are given in Chapter 2.

An application programmer, using the F-channel communication paradigm, has flexibility in defining the delivery order requirements. Defining the delivery order dynamically allows the programmer to choose the least amount of delivery order restrictions required, thus potentially improving the performance of the application. Unfortunately, the complexity of the system increases due to the additional nondeterminism in message passing. To help the user understand the system, we develop a methodology for reasoning about F-channel message passing in Chapter 3.

This thesis concerns the investigation of F-channel communication from the system's and user's perspectives. Using the irreflexive partial order defined intrinsically in F-channel communication, we explore implementation and verification areas for this non-traditional communication construct.

Let all things be done decently and in order.
New Testament, Corinthians XIV

Chapter 2

Implementation of a Flush Channel

Consider the logical unidirectional message path between two processes in a distributed system as shown in Figure 2.1. Let S denote the sending process; let R denote the receiver.

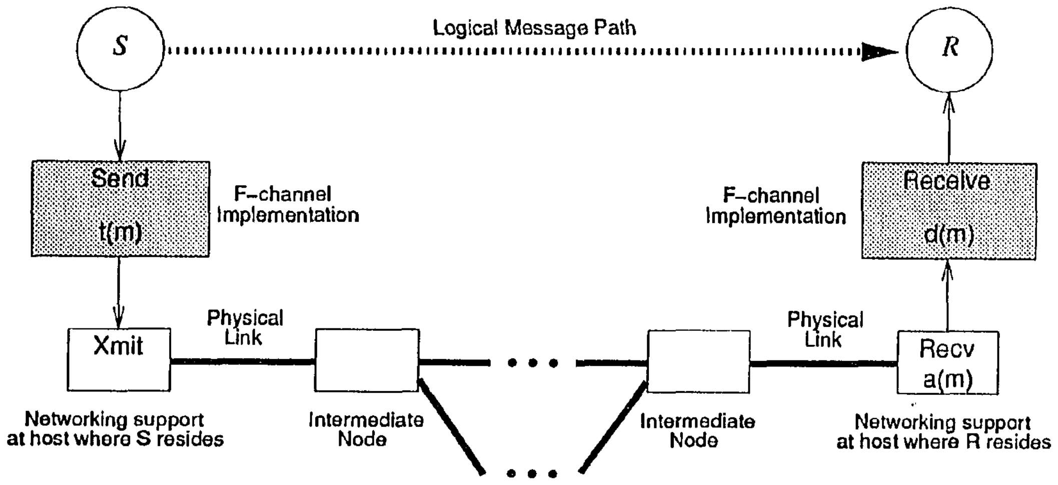

Figure 2.1: Logical and Physical Message Paths

In this network, one may identify several important events in the lifetime of a message m sent by S to R. $t(m)$ denotes the time of *transmission* of m by S; it is the time at which

m is passed to the networking support by S's execution of a **send** command. $a(m)$ is the time of m's *arrival* at the destination; m may be buffered at the destination node for some period of time. $d(m)$ is the time of m's ultimate *delivery* to the user process R—it is the time when the destination process can allow the receipt of m without violating any delivery order constraints.

In any communication paradigm (FIFO, unordered, and flush), a message may experience a delay at the destination host due to R's not having issued a **receive**. For both FIFO channels and F-channels, however, there may be an additional delay between a message's arrival and its delivery to the destination process. This further postponement is called *resequencing delay.* It is the interval $[a(m), d(m))$ in Figure 2.1, and it accrues due to differences between the arrival order and the allowed delivery order(s). For a FIFO channel, a given message cannot be delivered until all messages transmitted before it have been delivered. A message which takes a fast path through the underlying physical network and arrives early, out of order, must be buffered until all messages transmitted before it have arrived and been delivered. Resequencing delay is a major impediment in attempts to provide high bandwidth virtual circuits over multiple parallel links between source and destination (see, for example, [YN86, Cho89, AR87]).

The resequencing *problem* is more complex for F-channels because the (partial) delivery order required by an F-channel is generally more complex than the (total) FIFO order imposed by a virtual circuit or a sequenced packet channel. Intuitively, however, the resequencing *delay* for an F-channel is generally less than that for a virtual circuit. We expect the resequencing delay for an F-channel to increase when we add more restrictions to the delivery order. The delay, however, should approach that of a virtual circuit only in the worst case. As such, F-channels offer promise as a means of providing extremely high bandwidth inter-process communication over multiple transmission paths, without the programming disadvantages of datagram communication. Delivery order requirements (and, indirectly, the associated resequencing delays) are imposed by the programmer in keeping with the semantics of the distributed application.

We assume the existence of an effective network layer mechanism which assures reliable transmission of messages. We do not make any assumptions about transmission delay, arrival order, or routing policies. The network support provides the following operations:

Xmit data **to** dest

sends a message, with contents *data*, from the invoker's site to the site specified as *dest*. Once the message is passed to the network layer software (i.e., before its delivery to *dest*), the invoker continues.

Recv buff **from** src

will result in the delay of the invoker until a message arrives from host *src*. When such a message arrives, the contents of the message are stored in the invoker's address space at a location denoted by *buff*. The invoker then continues.

Reiterating our previous definitions, we use the term *arrival* to mean that a message has been received by the network support software at the destination host. The term *delivery* refers to a message's reception by the user process, the ultimate destination of the F-channel.

In summary, reexamining Figure 2.1, we need to implement the protocol layer which is shaded. We must provide users with F-channel **send** and **receive** operations which are faithful to F-channel message type semantics. Our minimal networking needs are met by the **Xmit** and **Recv** operations—we implement a layer of software which provides F-channel **send** and **receive** operations to user processes on top of this networking support [KC91, KCA92]. The guiding principle for the implementation of the **send** and **receive** primitives is based on *Pred*(m), the predecessor set of message m, defined in Section 1.2.

Flush Channel Implementation Policy:

Message m cannot be delivered unless all elements of *Pred*(m) have been delivered.

2.1 The "WaitFor" Technique

Consider *Pred*(m). This predecessor set includes all messages that must be delivered before message m can become eligible for delivery. To implement an F-channel, we deduce *Pred*(m) at the receiver by the following method. We augment each message at the transmitter with two integers: a unique *sequence number* and its *waitfor* value. The *waitfor* value is the highest sequence number in the message's predecessor set. The two integers effectively allow the destination to deduce the structure of the partial order and, hence, the delivery order restrictions imposed on the F-channel by the sender. Suppose that F-channel F connects a user process at site P_i with a user process at site P_j. As will be developed later in this section, an F-channel **send** for any of the four message types will ultimately result in the transmission of a message via a network call of the form

Xmit <*m.type*, *m.seqno*, *m.waitfor*, *m.data*> **to** P_j,

where *m.type* is m's flush message type, *m.seqno* is the sequence number of m, *m.waitfor* is m's *waitfor* value, and *m.data* is the data to be transmitted.

The basic idea behind the WaitFor technique is the observation that different message types have different criteria for delivery [KC91, KCA92]. How the sender sets the value of the *waitfor* field in a transmitted message and how the receiver interprets that value on arrival of the message are key in our adherence to the F-channel implementation policy. A two-way flush or a forward flush may not be delivered until all messages transmitted before it have also been delivered. Therefore, the sender sets *m.waitfor* to one less than *m.seqno*. The receiver infers that all messages with sequence numbers up to the *m.waitfor* value in a two-way flush or forward flush message must be delivered before m can be delivered. An ordinary message or a backward flush must wait only for the delivery of its immediate predecessor in $\lhd^{+}{}_{F}$. Therefore, *m.waitfor* is set by the sender to the sequence number of the current *backward flush point*. (The backward flush point is the last message transmitted that flushed the channel in a backward direction, i.e., the last two-way flush or backward flush message transmitted.) The receiver understands that an ordinary message or backward flush can be delivered when the message with sequence number *m.waitfor* has been delivered. The following discussion and pseudo-code show the actual details of the WaitFor technique.

The F-channel protocol at P_i must maintain two integers, in support of F, to determine the sequence number and *waitfor* value of each transmitted message. **seqno**(F) represents the sequence number of the last message transmitted over F-channel F. **bfp**(F) represents the sequence number of the last two-way flush or backward flush transmitted on F. Both **seqno**(F) and **bfp**(F) take their value from the set $\{-1, 0, 1, \ldots\}$ and are initially -1.

To send an ordinary message over F, the sender implements the following protocol:

```
Send (Ord, data) on $F \equiv$
    seqno($F$) := seqno($F$) + 1;
    Xmit <Ord, seqno($F$), bfp($F$), $data$> to $P_j$;
```

Sending a backward flush is implemented as:

```
Send (BF, data) on $F \equiv$
    seqno($F$) := seqno($F$) + 1;
    Xmit <BF, seqno($F$), bfp($F$), $data$> to $P_j$;
    bfp($F$) := seqno($F$);
```

A forward flush is implemented as:

```
Send (FF, data) on $F \equiv$
    seqno($F$) := seqno($F$) + 1;
    Xmit <FF, seqno($F$), seqno($F$) - 1, $data$> to $P_j$;
```

Finally, the two-way flush combines aspects of both the forward and backward flush types:

```
Send (2F, data) on $F \equiv$
  seqno($F$) := seqno($F$) + 1;
  Xmit <2F, seqno($F$), seqno($F$) - 1, $data$> to $P_j$;
  bfp($F$) := seqno($F$);
```

Upon the arrival of a message at P_j, the receiving flush channel implementation decides if it must buffer the message or make it eligible for delivery to the destination process. This decision is based on the interpretation of the *waitfor* field. A message arrives at the F-channel implementation of the receiving host P_j as a result of:

Recv <*m.type*, *m.seqno*, *m.waitfor*, *m.data*> **from** P_i.

In order to deduce whether this newly arrived message is deliverable, information about messages that have previously been delivered must be maintained. **delv**(F) is a set containing the sequence numbers of all messages which have arrived and have been delivered at site P_j. A message must be buffered if it arrives before the message(s) which makes it eligible for delivery; the set **buffer**(F) contains those messages which are currently buffered at the receiver. The time that a message spends in **buffer**(F) is the resequencing delay for that message.

We model the receiver as a daemon process which **Recv**s messages from the network, interprets them as messages on an F-channel, and deals with them appropriately. If the message arrives "too early" it must be buffered; otherwise it is delivered. The delivery of any message, newly arrived or formerly buffered, causes a re-examination of all buffered messages to see if any others are eligible for delivery. Below we list the pseudo-code for the implementation at the destination of an F-channel:

```
Process F–Daemon($P_i$ : site, $F$: FchannelID)
  while true do
    Recv <$m.type$,$m.seqno$,$m.waitfor$,$m.data$> from $P_i$;
    buffer($F$) := buffer($F$) $\cup$ {<$m.type$, $m.seqno$, $m.waitfor$, $m.data$>};
    change := true;
    while change do
      change := false;
      foreach $\mu \in$ buffer($F$) do
        DealWith($\mu$);
        if $\mu.seqno \in$ delv($F$) then
          change := true;
        fi
      od
    od
  od
end F–Daemon
```

The real decision concerning whether a message is eligible for delivery takes place in the routine DealWith. **Pipe(F)** is a FIFO buffer between the F-Daemon and the user process, R, as they stand in a producer/consumer relationship. The **Deposit** routine inserts message m, structured as an F-channel message (<type, data>), into **Pipe(F)** and handles all required synchronization. As far as we are concerned, **Depositing** a message in **Pipe(F)** is the delivery of the message; the destination process must issue a **receive** in order to **Consume** the message from the pipe.

```
Procedure DealWith($m$ : message)
  if ($m.type$ = Ord $\vee$ $m.type$ = BF) then
    if ($m.waitfor \in$ delv($F$) $\vee$ $m.waitfor = -1$) then
      buffer($F$) := buffer($F$) − {$m$};
      delv($F$) := delv($F$) $\cup$ {$m.seqno$};
      Deposit(<$m.type$, $m.data$>, Pipe($F$));
    fi
  else
    if ($\forall b : 0 \leq b \leq m.waitfor : b \in$ delv($F$)) then
      buffer($F$) := buffer($F$) − {$m$};
      delv($F$) := delv($F$) $\cup$ {$m.seqno$};
      Deposit (<$m.type$, $m.data$>, Pipe($F$));
```

```
            fi
        fi
    end DealWith
```

The implementation of the F-channel **receive** is reduced to removing a message from **Pipe**(F). We assume the existence of a routine, **Consume**, which performs all necessary producer/consumer synchronization and returns the message in **Pipe**(F) which was produced (**Deposit**ed into **Pipe**(F)) earlier than any others. Hence,

```
Receive (type, data) from $F \equiv$
    Consume(<t,d>,Pipe($F$));
    type, data := t, d;
```

Note that the synchronous nature of the F-channel **receive** operation follows naturally from the producer/consumer synchronization implemented on **Pipe**(F). An empty **Pipe**(F) delays a **receive**.

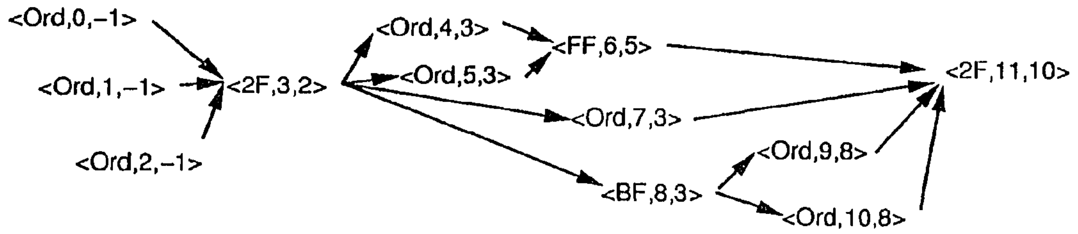

Figure 2.2: The Immediate Predecessor Graph in Terms of the WaitFor Protocol

Figure 2.2 shows the same covering relation as Figure 1.3, but here each message is augmented with the value of the *waitfor* field. The data field is omitted. The message labeled <Ord,1,-1> may be delivered as soon as it is received. It has no predecessors in the partial order. The two-way flush <2F,3,2> cannot be delivered until the first three ordinary messages have been delivered. Further, all messages to the right of <2F,3,2> will (explicitly or implicitly) not be delivered until <2F,3,2> has been delivered. The case of <Ord,10,8> is interesting—according to the protocol, as soon as the message with sequence number 8 is delivered, <Ord,10,8> may be delivered. But before <BF,8,3> may be delivered, <2F,3,2> must be delivered. Clearly the use of the *waitfor* field in the protocol takes into account the transitivity of $\lhd\!+_F$.

2.2 Correctness of the WaitFor Technique

Our argument in support of the correctness of the protocol is based upon the relationship between structural properties of the partial order and the components of the protocol. We ultimately want to establish that the F-channel protocols at the transmitter and receiver cooperate in such a way that the F-channel implementation policy is obeyed. An easy, but useful, first step in the argument is given by Lemma 1.

Lemma 1 *For a two-way flush or a forward flush, m,*

$$Pred(m) = \{m' : 0 \leq m'.seqno \leq m.waitfor\}.$$

Proof: All messages transmitted before m must have sequence numbers lower than $m.seqno$ as the sending F-channel software generates a monotonic stream of sequence numbers with an increment of unity. No message transmitted before m may be delivered after m. Hence, the predecessor set of m contains exactly those messages which were transmitted before m on the F-channel. The lemma follows from the fact that the sender sets $m.waitfor = m.seqno - 1$ for both two-way and forward flush messages. ∎

The case of ordinary and backward flushes is somewhat more complex. We begin with a lemma which establishes how these message types fit within $\lhd\!+_F$ by considering their immediate predecessors.

Lemma 2 *If m is an ordinary message or a backward flush and there exists a message m' such that $m'.seqno = m.waitfor$, then*

- $m' \lhd_F m$,
- *m' must be a two-way flush or a backward flush,*
- *there is no message $m'' \neq m'$ for which $m'' \lhd_F m$.*

Proof: On the transmission of m, an ordinary or backward flush message, the sender sets *m.waitfor* to the sequence number of the backward flush point. If m' is the backward flush point at the transmission of m, then the pair (m', m) is added to $\lhd_F$. By definition, the backward flush point is either empty or a singleton. If it is a singleton, then it is either a two-way flush or a backward flush message because forward flush and ordinary messages never alter the backward flush point.

Viewing $\lhd_F$ as an acyclic digraph, whenever a message is inserted into the covering relation, its in-degree is established, and that in-degree is not altered thereafter. Hence, m will have in-degree of one, $m' \lhd_F m$, and there can be no $m'' \neq m'$ such that $m'' \lhd_F m$. ∎

Having shown that an ordinary message or a backward flush has a unique predecessor in $\lhd_F$, if it has a predecessor at all, we establish exploitable structural properties of the entire predecessor set of such a message.

For an ordinary message or backward flush m, define the *BFP-chain* of m as the set of messages

$$\text{chain}(m) = \{m_k, m_{k-1}, \ldots, m_1\},$$

where

$$m_k \lhd_F m_{k-1} \lhd_F \ldots \lhd_F m_1 \lhd_F m.$$

Each element of chain(m) must be a backward or two-way flush. In graphical terms, the BFP-chain is the path of backward flushes back to the closest two-way flush, including that two-way flush. If there is no such two-way flush, it is the path of backward flushes back to the minimal backward flush in $\lhd +_F$. That is, m_k is a backward flush only if m has no two-way flush predecessor in $\lhd +_F$. The other m_i, $1 \leq i \leq k-1$ are backward flushes. *Head*(chain(m)), the head of the BFP-chain of m, is m_k, the two-way or backward flush which begins the chain of backward flushes leading to m.

Given this notation, we may be more precise in describing the complete predecessor set of ordinary and backward flushes.

Lemma 3 *If m is an ordinary message or a backward flush, then $m' \in$ Pred(m) if and only if $m' \in$* chain(m) $\cup$ *Pred*(*Head*(chain(m))).

Proof: [If] This part of the proof is straightforward. If $m' \in$ chain(m), then clearly $m' \in$ *Pred*(m). If $m' \in$ *Pred*(*Head*(chain(m))), then the transitivity of $\lhd +_F$ allows us to conclude that $m' \in$ *Pred*(m) since *Head*(chain(m)) must be a two-way flush for m' to exist.

[Only If] We proceed by contradiction. That is, suppose that some message $m' \in$ *Pred*(m), but $m' \notin$ chain(m) $\cup$ *Pred*(*Head*(chain(m))). This means that $\lhd_F$ must appear as shown in Figure 2.3. m' must be linked to m or some element of chain(m) $-$ *Head*(chain(m)), perhaps

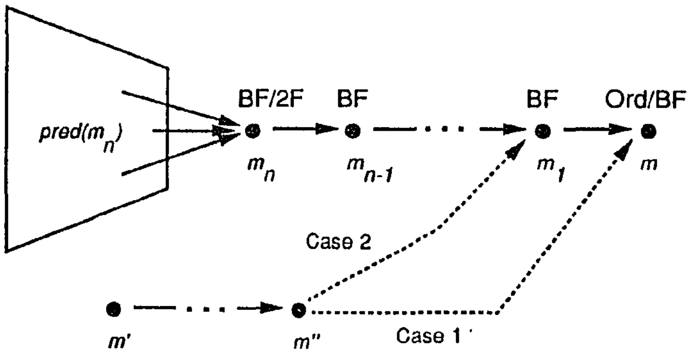

Figure 2.3: Contradiction: A Non-Chained Predecessor of m

through some successor in $\lhd_F$, due to the hypothesis that $m' \in$ *Pred*(m). That successor is denoted m'' in the figure. The first case is that $m'' \lhd_F m$, but Lemma 2 disallows the possibility of m having more than a single predecessor. In the second case, without loss of generality, assume that $m'' \lhd_F m_1 \in$ chain(m) $-$ *Head*(chain(m)). But by definition of a

BFP-chain, m_1 must be a backward flush. Again, Lemma 2 disallows this edge in $\lhd_F$. In both cases we have reached a contradiction. ∎

Having presented the above useful lemmas, we may now prove that the WaitFor protocol faithfully implements the Flush Channel Implementation Policy.

Theorem 1 (SAFETY) *Under the WaitFor technique, message m is consumed by a* **receive** *at the destination process only if Pred(m) has already been consumed.*

Proof: In DealWith, if m is a two-way flush or forward flush, it will not be **Deposit**ed in **Pipe**(F) unless all messages with equal or lower sequence numbers have been **Deposited**. By Lemma 1, these messages are precisely the predecessor set of m.

For an incoming backward flush or ordinary message m, Lemma 2 shows that *m.waitfor* is the sequence number of the immediate predecessor of m. In the definition of chain(m), m_1 is that immediate predecessor. DealWith will not allow the delivery of m until after the delivery of m_1. Generalizing this argument to each backward flush on the BFP-chain of m, DealWith will insist the **Deposits** are correctly ordered. The correctly ordered **Deposit** of *Head*(chain(m)) and of *Pred*(*Head*(chain(m))) are handled by the protocol for two-way flushes, which was shown to be correct in the first part of this proof. We therefore conclude, by Lemma 3, that *Pred*(m) must have been delivered before m is delivered.

As a final part of this argument, it is essential that **Pipe**(F) be a FIFO buffer. Thus, user-invoked **receives** will **Consume** messages in the same order in which the protocol recognizes that they are eligible for delivery and **Deposits** them in **Pipe**(F). ∎

Having proved the WaitFor protocol adheres to the Flush Channel Implementation Policy, one last step in our argument for correctness is necessary. A liveness proof ensures that something *good* will eventually happen during execution [AS85, OL82].

Theorem 2 (LIVENESS) *Assume that the receiver will, in fact, issue a* **receive** *for each* **send** *executed by the sender. Message m, sent on an F-channel implemented as described above, will then be received in finite time.*

Proof: Since the network is assumed to be reliable, a message must arrive at the destination host within finite time from its transmission, and therefore the F-Daemon will **Recv** the message. We proceed by induction on message sequence number. Our basis is the case that $m.seqno = 0$. In this case $m.waitfor = -1$, and thus m will be **Deposited** in **Pipe**(F) without delay. The first **receive** will therefore **Consume** m within a finite time from its transmission.

Assume that messages with sequence numbers up to and including n will be **Consumed** within finite time. Consider the case where $m.seqno = n+1$. By Theorem 1, m will not be **Deposited** in **Pipe**(F) until its predecessor set has been **Deposited**. Its predecessor set will include messages with sequence numbers no greater than n, since $m.waitfor < m.seqno$, $m.waitfor$ is defined to be the highest sequence number in $Pred(m)$, and $m.seqno = n + 1$. By the inductive hypothesis, we conclude that $Pred(m)$ will be **Deposited** and **Consumed** in finite time. m will then be **Deposited** by DealWith as invoked by the F-Daemon. The FIFO nature of **Pipe**(F) then implies that m will eventually be **received**. ∎

2.3 Previous Implementation Techniques

Two implementations for F-channels have appeared in the literature. The following sections review these techniques.

2.3.1 The Flooding Protocol

The first implementation in the literature [Ahu91] adheres to the F-channel implementation policy by flooding each physical network path between sender and receiver for all but ordinary messages. The technique assumes a reliable network with every switch node in the network having incoming and outgoing FIFO queues for the incoming and outgoing channels connected to the node (see Figure 2.4). A flush message (that is, a message of

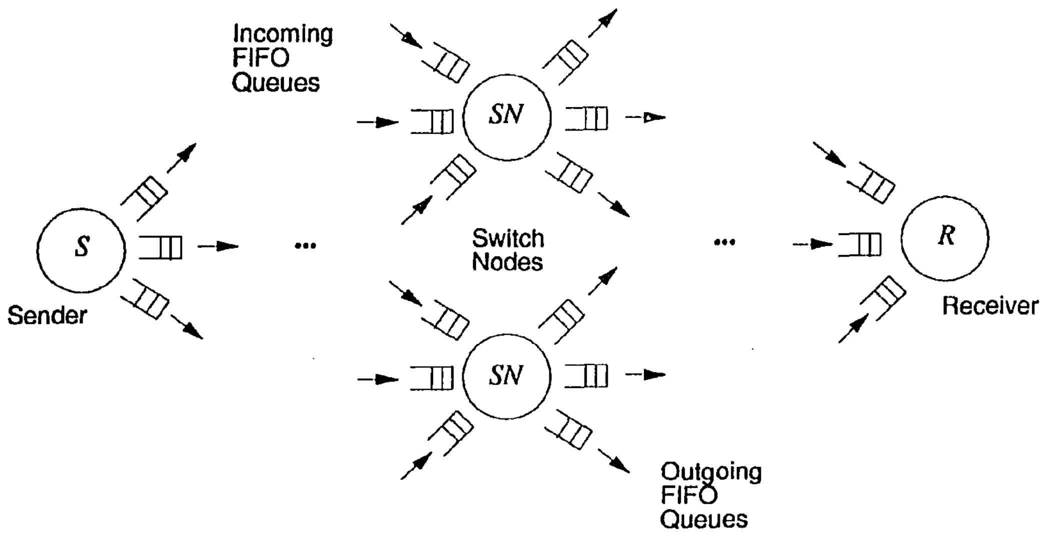

Figure 2.4: The Reliable Network in the Flooding Protocol

type **2F**, **FF**, or **BF**) is transmitted at the sender by placing a copy of the message in all outgoing queues that lead to the destination. When a copy of a flush message arrives at a switch node, the node places a copy of the message in all outgoing queues that lead to the destination. Thus, a copy of the flush message is transmitted over every network channel between sender and receiver. A single copy of an ordinary message is routed from S to R through some path in the underlying physical network.

A two-way flush is made eligible for delivery at R when every incoming FIFO queue has the two-way flush at the head of the queue. Any incoming queue with the two-way flush at the head of the queue is blocked until every incoming queue has the two-way flush at the head of the queue. This guarantees that the two-way flush will be delivered after every message transmitted before it and before every message transmitted after it.

Likewise, a forward flush is made eligible for delivery when a copy of the message is at the head of each incoming queue. Thus, a forward flush will be delivered after every message transmitted before it. In this situation, however, the incoming queues that have seen the forward flush are not blocked—messages transmitted after the forward flush may be delivered while the forward flush waits for its delivery.

At R, a backward flush is made eligible for delivery at the arrival of the first copy of the backward flush. As in the forward flush case, the incoming queues are not blocked. Handling a backward flush in this manner guarantees delivery of the backward flush before every message transmitted after it.

Nodes in the network pass ordinary messages through without any delays, blockages, or copying. The only delivery delays imposed on an ordinary message are those imposed by the other three message types.

The Flooding Protocol describes selective flooding of all network paths between sender and receiver for all but ordinary messages. In comparison, the WaitFor technique requires only a single copy of each message; the sequence number and the *waitfor* value are piggybacked on the message, thus allowing the receiver to deduce a message's place in the partial order.

2.3.2 The Three Counter Technique

In this section, we introduce the second implementation for F-channels available in the literature. We term this implementation the Three Counter technique and copy it from its original presentation [AVS91]. An F-channel between processes p and q is denoted $c_{p,q}$.

> We presume that for any $c_{p,q}$, $p(q)$ has an out-buffer (in-buffer) in which p puts (from which q takes) messages to be sent (received) along $c_{p,q}$. In the following, all messages referred to are those sent (received) along $c_{p,q}$. The implementation protocol for $c_{p,q}$ is as follows:
>
> p has counters T and M. T has a value equal to 1 plus the number of *two-way-flushes* sent so far. M has a value equal to 1 plus the number of messages sent so far after the latest *two-way-flush* sent.[1]

[1]Initially, T and M are each 1. After sending a *two-way-flush* and before sending any other messages, T is incremented by 1 and M is reinitialized to 1. After sending a message other than *two-way-flush* and before sending any other messages, M is incremented by 1.

p assigns message m (to be sent) an identity $<Type, T_m, M_m>$, where T_m and M_m are values of T and M when m is sent and $Type$ is the message type, ordinary, *two-way-flush*, *forward-flush*, or *backward-flush*.

q initially assumes that it has received a *two-way-flush* with identity $<two - way - flush, 0, 0>$; thus the *two-way-flush* that we will refer to as the ${T_m}^{th}$ *two-way-flush* will have identity $<two - way - flush, (T_m - 1), *>$ where $*$ can be any value.

q receives m that is a *two-way-flush* or a *forward-flush* from the input buffer only after it has received the ${T_m}^{th}$ *two-way-flush* and each message m' with $M_{m'}$ less than M_m (and more than 0) since receiving the ${T_m}^{th}$ *two-way-flush.*

If before sending m that is either an ordinary message or a *backward-flush* and after sending the ${T_m}^{th}$ *two-way-flush*, – has sent one or more *backward-flushes*, then along with m p sends M_{bf} where bf is the latest *backward-flush* sent before m.[2]

q receives m that is either an ordinary message or a *backward-flush* from the input buffer only after the ${T_m}^{th}$ *two-way-flush* and the *backward-flush* with identity $<backward - flush, T_m, M_{bf}>$, if M_{bf} was carried with m [AVS91].

The Three Counter technique resembles the WaitFor technique in that integers convey receipt order information. The technique, however, was never formally shown to be correct. Therefore, one is skeptical that the protocol actually adheres to the F-channel implementation policy. In the next section, we show that the WaitFor technique and the Three Counter technique are functionally equivalent. Hence, the correctness of the WaitFor technique also serve as a proof of the validity of the Three Counter technique.

Another criticism of the Three Counter technique is that it uses only two of the three required fields at the destination process to decide whether a message is ready for delivery—each message transmitted on the F-channel stores a value that is never used. Furthermore, the Three Counter technique, at the delivery of a backward flush or ordinary message, ensures the previous two-way flush message has been delivered. Checking for the delivery of the previous two-way flush is redundant if a previous backward flush, transmitted after the two-way flush, exists. Due to transitivity in the partial order, the previous two-way flush is guaranteed delivery if the previous backward flush is delivered.

[2]That is, rather than sending just $<Type, T_m, M_m>$ with m, p send $<Type, T_m, M_m, M_{bf}>$. Note that T_{bf} must be the same as T_m.

2.4 Three Counter Technique ≡ WaitFor Technique

Both the Three Counter technique and the WaitFor technique use integers appended to a message to convey delivery order information. We now prove that the two techniques are functionally equivalent. That is, given an arbitrary sequence of messages transmitted on an F-channel and an arbitrary arrival order, both techniques generate the same delivery order. Once the equivalence is shown, the correctness of the WaitFor technique in Section 2.2 serves as the missing validity proof for the Three Counter technique.

As before, suppose that F-channel F connects a user process at site P_i with a user process at site P_j. To convey receipt order information in the Three Counter technique, the sender augments a message with the three fields described in Section 2.3.2. T_m is the number of the next two-way flush that will be transmitted. C_m is the number of messages that have been transmitted since the transmission of the last two-way flush. C_{bf} is zero if no backward flush messages have been transmitted since the transmission of the last two-way flush; otherwise, C_{bf} becomes the value of C_m after the transmission of the backward flush.

The Three Counter technique relies on the structure of the partial order between two-way flush messages. We define *prev*2F, at any given time, as the last two-way flush message transmitted and *nth*2F as the *nth* two-way flush message transmitted.

Lemma 4 *For a two-way flush or a forward flush message, m,*

$$Pred(m) = Pred(prev\mathbf{2F}) \cup \{m' : prev\mathbf{2F}.seqno \leq m'.seqno \leq m.waitfor\}.$$

Proof: The predecessor set of a two-way flush or forward flush message, m, contains exactly those messages which were transmitted before m on the F-channel (Lemma 1). If no previously transmitted two-way flush exists, then *Pred(prev*2F) is empty and $prev\mathbf{2F}.seqno = 0$. If a previously transmitted two-way flush does exist, then, by Lemma 1,

$$Pred(prev\mathbf{2F}) = \{m' : 0 \leq m'.seqno \leq prev\mathbf{2F}.waitfor\}.$$

Since $prev\mathbf{2F}.seqno = prev\mathbf{2F}.waitfor + 1$,

$$\begin{aligned} &Pred(prev\mathbf{2F}) \cup \{m' : prev\mathbf{2F}.seqno \leq m'.seqno \leq m.waitfor\}, \\ &= \{m' : 0 \leq m'.seqno \leq m.waitfor\} \\ &= Pred(m). \end{aligned}$$ ∎

Lemma 5 *For a two-way flush or a forward flush message, m, if all messages transmitted between the $(T_m - 1)th$ two-way flush (inclusive) and the $C_m th$ message have been delivered in the Three Counter technique, then all messages in $Pred(m)$ have been delivered.*

Proof: Suppose m is a two-way flush or forward flush message and all messages transmitted between the $(T_m - 1)th$ two-way flush (inclusive) and the $C_m th$ message have been delivered. By Lemma 4, $Pred(m)$ is all messages transmitted between the $(T_m - 1)th$ two-way flush (inclusive) and the $C_m th$ message and the predecessor set of the $(T_m - 1)th$ two-way flush. To verify $Pred(prev\mathbf{2F})$ is delivered before m, $prev\mathbf{2F} = (T_m - 1)th$ two-way flush, we proceed by induction on the number of two-way flush messages transmitted before m.

The basis case is that no two-way flush messages have been transmitted before m. Then $Pred(prev\mathbf{2F})$ is empty. Assume that n two-way flush have been transmitted before m and all messages in $Pred(prev\mathbf{2F})$, $prev\mathbf{2F} = nth\mathbf{2F}$, are delivered before m is delivered. Consider the case where $(n + 1)$ two-way flush messages are transmitted before m: $prev\mathbf{2F} = (n + 1)st\mathbf{2F}$. By Lemma 4, $Pred((n + 1)st\mathbf{2F})$ is the predecessor set of the nth two-way flush and all messages transmitted between the nth two-way flush (inclusive) and m. It is given that the previous two-way flush, the $(n + 1)st\mathbf{2F}$, has been delivered. In the Three Counter technique, this $(n + 1)st\mathbf{2F}$ is delivered only if all messages transmitted between the nth two-way flush (inclusive) and the $(n + 1)st$ two-way flush have been delivered as well. By the inductive hypothesis, $Pred(nth\mathbf{2F})$ has been delivered before m, hence $Pred(prev\mathbf{2F})$, where $prev\mathbf{2F} = (n + 1)st\mathbf{2F}$, is delivered before m is delivered. Since $Pred(m)$ is the set of all previously transmitted messages (Lemma 1), all messages transmitted before m have been delivered. ∎

Lemma 6 *Suppose m is a backward flush or ordinary message transmitted on an F-channel implemented with the Three Counter technique. Pred(m) has been delivered if the $(T_m - 1)th$ two-way flush is delivered and $C_{bf} = 0$ or the $(T_m - 1)th$ two-way flush and the backward flush with C_m equal to C_{bf} have been delivered.*

Proof: Let m be a backward flush or ordinary message. If $C_{bf} = 0$, then no backward flush has been transmitted since *prev*2F, the $(T_m - 1)th$ two-way flush. Thus, *prev*2F is the backward flush point at the transmission of m. If $C_{bf} \neq 0$, then C_{bf} is the counter of the last backward flush transmitted. Again, this message is the backward flush point at the transmission of m. In the Three Counter protocol, m is delivered once the $(T_m - 1)th$ two-way flush is delivered and the message with counter C_{bf} (unless equal to zero) is delivered. In summary, m is delivered if the backward flush point of m is delivered. By an identical argument in Theorem 1, we conclude that *Pred*(m) is delivered if m's backward flush point is delivered. ∎

Theorem 3 *Given the same partial order and the same arrival order at the destination, the WaitFor technique and the Three Counter technique generate the same delivery order.*

Proof: Suppose two F-channels are available. One F-channel implements the WaitFor protocol; the other F-channel implements the Three Counter technique. Both F-channel implementations are given the same partial order and the same arrival order. By Lemmas 5 and 6, m in the Three Counter protocol is made eligible for delivery when *Pred*(m) is delivered. Likewise, in the WaitFor technique and proven in Theorem 1, m is **Deposit**ed by DealWith only if *Pred*(m) has been **Deposit**ed. Therefore, the eligibility of a message is based on the same factors in both protocols. ∎

Now that the equivalence has been shown, we display the Three Counter technique in a manner similar to the presentation of the WaitFor technique. The **send** operation for this new protocol produces a network call of the form

$$\textbf{Xmit} <type_m,\ T_m,\ C_m,\ C_{bf},\ data_m> \textbf{ to } P_j,$$

where T_m, C_m, and C_{bf} are the three integers that convey delivery order information to P_j and $data_m$ is the data to be transmitted.

The F-channel implementation at site P_i must maintain three integers, in support of F, to determine T_m, C_m, and C_{bf}. We represent these integers as **T**(F), **C**(F), and **CBF**(F). Initially, the values of **T**(F), **C**(F), and **CBF**(F) are 1, 1, and 0 respectively.

To send an ordinary message over F, the sending F-channel in the Three Counter technique implements the following protocol:

```
Send (Ord, data) on $F \equiv$
    Xmit <Ord, T($F$), C($F$), CBF($F$), $data$> to $P_j$;
    C($F$) := C($F$) + 1;
```

Sending a backward flush message is implemented as:

```
Send (BF, data) on $F \equiv$
    Xmit <BF, T($F$), C($F$), CBF($F$), $data$> to $P_j$;
    CBF($F$) := C($F$);
    C($F$) := C($F$) + 1;
```

A forward flush is implemented as:

```
Send (FF, data) on $F \equiv$
    Xmit <FF, T($F$), C($F$), CBF($F$), $data$> to $P_j$;
    C($F$) := C($F$) + 1;
```

Finally, the two-way flush is implemented as:

```
Send (2F, data) on $F \equiv$
    Xmit <2F, T($F$), C($F$), CBF($F$), $data$> to $P_j$;
    T($F$) := T($F$) + 1;
    C($F$) := 1;
    CBF($F$) := 0;
```

Upon the arrival of a message at P_j, the receiving F-channel decides if it must buffer the message or make it eligible for delivery to the destination process. A message arrives at the receiving host P_j as a result of:

Recv <$type_m$, T_m, C_m, C_{bf}, $data_m$> **from** P_i.

The receiver maintains three data structures in order to deduce the delivery order. **last2F**(F) is an integer, initially zero, that represents the number of previous two-way flush messages that have been delivered. **counter**(F) is a set, initially empty, that contains the C_m fields of all the messages that have been delivered since the delivery of the last two-way flush. **buffer**(F), as in the WaitFor technique, is a set that contains those messages which are currently not eligible for delivery. We model the receiver in the Three Counter technique, similar to the WaitFor technique, as a daemon process which receives messages, interprets them, and deals with them appropriately.

```
Process F–Daemon($P_i$ : site, $F$: FchannelID)
  while true do
    Recv <$type_m$, $T_m$, $C_m$, $C_{bf}$, $data_m$> from $P_i$;
    buffer($F$) := buffer($F$) $\cup$ {<$type_m$, $T_m$, $C_m$, $C_{bf}$, $data_m$>};
    change := true;
    while change do
      change := false;
      foreach $\mu \in$ buffer($F$) do
        DealWith($\mu$);
        if $\mu \notin$ buffer($F$) then
          change := true;
        fi
      od
    od
  od
end F–Daemon
```

As in the WaitFor technique, the real decision making takes place in DealWith.

```
Procedure DealWith($m$ : message)
  if ($type_m$ = Ord $\vee$ $type_m$ = BF) then
    if ((last2F($F$) = $T_m - 1$) $\wedge$ ($C_{bf} = 0 \vee C_{bf} \in$ counter($F$))) then
      buffer($F$) := buffer($F$) $-$ $\{m\}$;
      counter($F$) := counter($F$) $\cup$ $\{C_m\}$;
      Deposit ($<type_m, data_m>$, Pipe($F$));
    fi
  else
    if ((last2F($F$) = $T_m - 1$) $\wedge$ ($\forall b : 1 \leq b < C_m : b \in$ counter($F$))) then
      buffer($F$) := buffer($F$) $-$ $\{m\}$;
      if ($type_m$ = 2F) then
        last2F($F$) := $T_m$;
        counter($F$) := $\{\}$;
      else
        counter($F$) := counter($F$) $\cup$ $\{C_m\}$;
      fi
      Deposit ($<type_m, data_m>$, Pipe($F$));
    fi
  fi
end DealWith
```

Figure 2.5 shows the same immediate predecessor graph as Figure 2.2, but here each message is augmented with the fields required in the Three Counter technique.

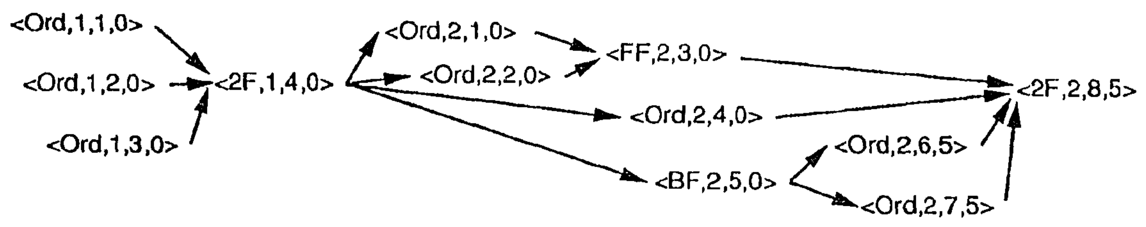

Figure 2.5: The Immediate Predecessor Graph in Terms of The Three Counter Technique

2.5 Flow Control Issues

Before a prototype of the WaitFor protocol can be practically implemented, significant issues related to buffer capacity and sequence numbers must be solved. We deal with these issues in this section.

2.5.1 Bounding Buffers

In most communication protocols, both the sending and receiving processes require message buffering capabilities. The sender's buffer stores each message until an acknowledgement (ACK), explicit or implicit, is returned. The receiver's buffer stores the messages that have arrived but are not ready for delivery. A buffered message at a receiver is attributed to either an arrival order inconsistent with the required delivery order or, simply, a receiving user process which is slow to **receive** messages. One issue considered here is the problem encountered due to bounded buffer space at both the sender and receiver. We assume, in the following discussion, that no messages are lost in transit, that buffer overflow at the receiver is the only cause for retransmission, and that a message is buffered at the sender until it is explicitly ACKed.

The difference in size between the sender's and receiver's buffers produces various effects. If the receiver's buffer is larger than the sender's buffer, then the excess buffer space at the destination process will never be used. If the receiver's buffer is equal in size to the sender's buffer, then buffer space is not wasted, and space is always available for arriving messages. In this case, retransmission of messages will never be required as there will be no buffer overflow at the receiver.

Retransmission of messages, however, might be required when the receiver's buffer is smaller than the sender's buffer. For a FIFO channel, it is obvious which message to retransmit upon a message's arrival at a full receiving buffer. The delivery order for a FIFO channel is identical to the transmission order; therefore, the sequence number at the receiver representing the *last* message transmitted is the message selected to be retransmitted. Selecting to retransmit the *last* message transmitted delays the delivery of the fewest messages—all messages in the buffer must be received before this *last* message.

The choice of which message to retransmit at a full receiving buffer in an F-channel implementation is not as obvious. Upon buffer overflow, the receiver must decide upon a message to discard (thereby causing a retransmission) that will delay the delivery of the fewest messages. An F-channel implementation could use one of many different retransmission strategies: the first or last message transmitted, the type of flush message, the smallest message in the buffer, a random choice, etc. Of course, threats to liveness must be considered when devising a retransmission scheme.

Theorem 4 *Any retransmission strategy that handles a full receiving buffer in an F-channel implementation cannot be optimal at all times.*

Proof: An optimal retransmission strategy for a full buffer would always discard the message which yields the fewest (future) retransmissions. As the receiver has no knowledge of the full partial ordering on messages being transmitted or the order in which these messages will arrive, the best retransmission selection cannot always be made. For example, Figure 2.6 is an immediate predecessor graph that illustrates the possible impact of the receiving F-channel software not having perfect (future) knowledge. Suppose the sender's

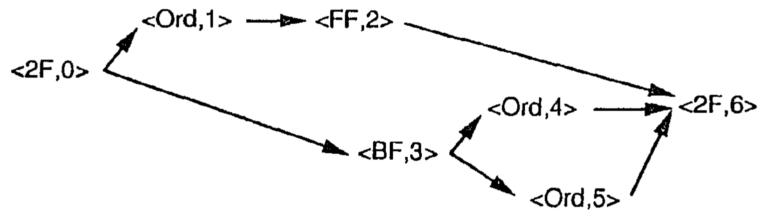

Figure 2.6: The Retransmission Problem

buffer is of size seven, thereby allowing the transmission of all seven messages in succession, while the receiving buffer is of size one. If messages with sequence numbers 1 and 3 are the first two arrivals, the optimal choice of which message to discard cannot be made. If the arrival order is 4 5 0 2 6 following the arrival of 1 and 3, the system would be best served by retransmitting message with sequence number 3. On the other hand, an arrival order of 0 4 5 2 6 produces fewer retransmissions if message 1 is discarded. Since a receiver will not have knowledge of the future arrival order, no retransmission strategy can always make the optimal selection for retransmission. ∎

Theorem 4 illustrates the difficulties associated with any retransmission strategy. We do not have a general solution for this problem. Instead we assume, throughout this thesis, the sizes of the sender's and receiver's buffers are equivalent. That is, at the establishment of an F-channel, we presuppose that a negotiation of an equivalent buffer size takes place between the sender and receiver. This assumption will forgo the necessity of deciding which message to discard at a full buffer.

2.5.2 Bounding Sequence Numbers

Bounding sequence numbers is trivial for any FIFO channel implementation, but not obvious for all F-channel implementations. The total delivery order of messages and the ACKing of each delivered message by the network support reveals a simple technique for bounding sequence numbers in a FIFO channel. The number of values required to distinguish each message in the channel is equivalent to the size of the sending buffer. As each message is ACKed, the sequence number of the delivered message is available for reuse.

In an F-channel implementation, bounding sequence numbers is not as simple. The difficulties appear when the sequence number of an F-channel message has a longer *lifetime* than the message itself. For example, in the flooding protocol of Section 2.3.1, sequence numbers only identify the message en route; thus a sequence number is available for reuse once the sender receives an ACK for the message's receipt. On the other hand, in the WaitFor protocol of Section 2.1, the *waitfor* field is the sequence number of a message previously transmitted. Therefore, the *waitfor* field may extend the lifetime of a sequence number. That is, we cannot conclude that the delivery of a message with sequence number x finalizes all references to x.

Theorem 5 *An F-channel that uses the WaitFor protocol cannot use the policy to bound sequence numbers employed in FIFO channels.*

Proof: In FIFO channels, a sequence number can be reused at the sender once the corresponding message has been ACKed by the receiver. In F-channel communication, violations could occur in the delivery order if the reuse of a sequence number was based only on the ACK of the corresponding message. To illustrate the problem, consider the delivery of a message in the WaitFor technique; the delivery is based on the type of message and its *waitfor* value. Now consider the immediate predecessor graph, augmented with the *waitfor* field, in Figure 2.7. Suppose the sender's buffer is of size two. Upon the ACK of <2F,0,-1> and <Ord,1,0>, the sender would reuse the sequence numbers by transmitting <BF,0,0> and the second <Ord,1,0>. The *waitfor* fields in both of these messages are zero; however, the zeros do not refer to the same message. The receiver has no means to distinguish between the two zeros, and thus a message (e.g. the second <Ord,1,0>) may be delivered before it is eligible. This would violate the F-channel implementation policy. ∎

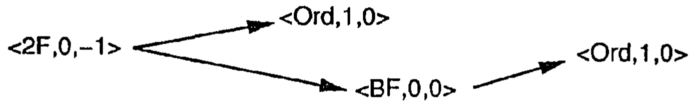

Figure 2.7: The Bounding Sequence Number Problem

Although, in the WaitFor protocol, a sequence number cannot be reused upon the ACK of the corresponding message, some numbers become available at an ACK of a two-way flush message due to the semantics of a two-way flush message.

Theorem 6 *In the WaitFor technique, if a message m is transmitted after a two-way flush message m', then*

$$m.waitfor \geq m'.seqno$$

Proof: Suppose message m is transmitted after m', a two-way flush message. Based on the semantics of a two-way flush message and the definition of a predecessor set, $m' \in Pred(m)$. Lemma 1 proved that all messages transmitted before m' will have sequence numbers lower than $m'.seqno$. Since $m.waitfor$ is the highest sequence number in $Pred(m)$, $m.waitfor \geq m'.seqno$. ∎

A solution to handle the fixed-width representation of sequence numbers in the WaitFor protocol is revealed by Theorem 6. Once an ACK of a two-way flush is received, all sequence numbers preceding the sequence number of the two-way flush can be reused. The following section modifies the WaitFor technique to handle fixed-width representation of sequence numbers. As will be shown, other flow control issues are a direct effect of bounding sequence numbers. To implement this protocol, the sending F-channel maintains the sequence number of the last two-way flush ACKed. At the receipt of an ACK for another two-way flush, the sender can reuse sequence numbers in the range [previous **2F** sequence number, current **2F** sequence number).

If the system transmits a group of messages with no two-way flush messages and the number of messages in the group is larger than the modulus used to bound sequence numbers, then the system must wait until every message is ACKed before reusing any number. That is, the system transmits a *dummy* two-way flush on the F-channel. Upon the delivery of the *dummy* two-way flush, the receiver resets all its variables to their initial values and transmits an ACK for the *dummy* message. Upon the receipt of this ACK, the sender resets all its variables as well; all sequence numbers then become available. Without synchronizing the system in this manner, bounding sequence numbers could violate the F-channel implementation policy (recall Theorem 5) or become deadlocked.

The use of a *dummy* two-way flush would not be required if the modulus for sequence numbers was based on additional knowledge: the maximum number of messages transmitted between any two consecutive two-way flush messages. Given this value, Theorem 7 illustrates a bound for sequence numbers.

Theorem 7 *Assume every F-channel application transmits a two-way flush as the first and last message of each application. In addition, assume max is the maximum number of messages between any two consecutive two-way flush messages and k is the size of the sender's (and receiver's) buffer. The maximum range of sequence numbers required to distinguish messages in an F-channel implementation is then* $[0, max + k)$.

Proof: Figure 2.8 illustrates an immediate predecessor graph that uses all the sequence numbers in the allowed range: $[0, max + k - 1)$. To verify no more numbers are required to distinguish the messages, we consider the situation of the F-channel preceding and following the transmission of the next message (augmented with sequence number 0). This example covers the worst possible scenario by having *max* messages transmitted between two consecutive two-way flush messages.

Consider the two consecutive two-way flush messages shown in the figure. If the sender transmits the message with sequence number $max + k - 1$, we prove that at most one unACKed message can exist between the two consecutive two-way flush messages. Without loss of generality, assume <Ord,1,0> is an unACKed message. No message with sequence number greater than *max* can then be delivered. Since all messages with sequence numbers between the range $[max + 1, max + k - 1]$ are unACKed, only one space in the sender's buffer is left for the unACKed <Ord,1,0> message. Once <Ord,1,0> is ACKed, the sending

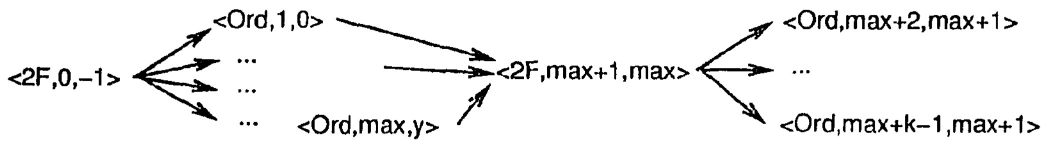

Figure 2.8: The Largest Sequence Number

F-channel software transmits the next message as sequence number 0. By Theorem 6, no message currently in the sending buffer can refer to a message with sequence number less than $(max + 1)$; reusing sequence number 0 will not violate the F-channel implementation policy. Hence, a sufficient modulus for sequence numbers in an F-channel is $max + k$. ∎

Theorem 7 provides a bound for sequence numbers given the value *max*. Obviously, knowledge of *max* is not always possible since prior knowledge of all messages that will be transmitted on the F-channel is unlikely. Therefore, an implementation of an F-channel with fixed-width representation for sequence numbers should be based only on the two-way flush effect of Theorem 6. In the following section, we modify the WaitFor protocol to include all the bounding considerations discussed in this section.

2.6 The Bounded WaitFor Technique

Suppose every F-channel application implicitly transmits a two-way flush message as the first and last message of each application. In addition, suppose *num* is the sequence number modulus used in the F-channel implementation. We assume $num > 1$; otherwise a FIFO delivery order occurs. One effect of bounded sequence numbers is that the size of the sending and receiving buffers becomes bounded as well—the size need not be larger than *num*. Buffer sizes of *num*, however, will generally be too large for the system. Therefore, in this protocol, we continue to assume the buffer size at both the sender and receiver is k.

Following the original protocol, the sending F-channel software sets the value of the *m.waitfor* field, and the receiving F-channel software interprets that value in order to adhere to the F-channel implementation policy. To support the F-channel at site P_i, the sender continues to maintain the two integers **seqno**(F) and **bfp**(F). In our bounded WaitFor technique, however, both **seqno**(F) and **bfp**(F) are initially 0 and take on values from the range $\{0, 1, \ldots, num - 1\}$. Another integer, maintained by the sender, is necessary to represent the last two-way flush message ACKed by the receiver: **2Fack**(F). Initially, **2Fack**(F) is 0. The sender's buffer, of maximum size k, is called **sbuffer**(F). A message is removed from **sbuffer**(F) once an ACK is received. We assume a timing mechanism retransmits a message if its ACK is not received within a given time. These messages are marked as *retransmitted*; we assume the destination process checks for previous delivery.

In the original protocol, messages are transmitted unconditionally. In the bounded protocol, a message cannot be transmitted unless space exists in the sender's buffer and a sequence number is available. Therefore, all transmissions are conditioned on the following criteria:

Transmission Condition:

$$|\mathbf{sbuffer}(F)| < k \wedge (\mathbf{seqno}(F) + 1) \mathbf{\ mod\ } num \neq \mathbf{2Fack}(F).$$

If the transmission condition is true, then the sender can transmit a message. If, however, the transmission condition is false, then the sender is delayed. We modify **2Fack**(F) in two situations. First, when the sender receives an ACK of a two-way flush message, **2Fack**(F) is set to the sequence number of the ACKed two-way flush. Deadlock would occur, however, if the sender received ACKs on every message in the buffer and no two-way flush message was transmitted in the last *num* messages. The second situation that modifies **2Fack**(F) covers this circumstance. If the sender's buffer is empty and the transmission condition continues to fail, the sender transmits a *dummy* two-way flush. The delivery of this *dummy* message resets all the variables at the receiver. Its corresponding ACK, in the same vein, resets all the variables at the sender. Updating the system in this manner re-initializes the system. As no messages are in transit, re-initialization under these conditions has no effect on the correctness of the protocol.

The following pseudo-code is the bounded F-channel implementation for the sender. We model the sender as a daemon process that either receives ACKs or transmits messages. Two binary flags in the process are assumed to be maintained outside the daemon. **ACK** is maintained by the network; when **ACK** is true, an ACK is available to be received. **MSG**(m) is maintained by software between the user application program and the daemon; when **MSG**(m) is true, the message <*m.type*, *m.data*> is ready to be transmitted.

```
Process FS-Daemon($P_j$ : site, $F$: FchannelID)
  while true do
    if (|sbuffer($F$)| $= k \vee$ ACK) then
      Recv <Ord, $ack.seqno, -1, \Phi$> from $P_j$;
      foreach $\mu \in$ sbuffer($F$) do
        if ($\mu.seqno = ack.seqno$) then
          $m.type$, $m.data := \mu.type$, $\mu.data$;
          $m.seqno$, $m.waitfor := \mu.seqno$, $\mu.waitfor$;
        fi
      od
      sbuffer($F$) := sbuffer($F$) $- \{m\}$;
      if ($m.seqno = -1$) then
        seqno($F$) := 0;
        bfp($F$) := 0;
        2Fack($F$) := 0;
      else
        if ($m.type =$ 2F) then
          2Fack($F$) := $m.seqno$;
        fi
      fi
    else
      if ((seqno($F$) + 1) mod $num \neq$ 2Fack($F$)) then
        if MSG($m$) then
          seqno($F$) := (seqno($F$) + 1) mod $num$;
          $m.seqno$ := seqno($F$);
          if ($m.type =$ Ord $\vee$ $m.type =$ BF) then
            $m.waitfor :=$ bfp($F$)
          else
            $m.waitfor :=$ (seqno($F$) $- 1 + num$) mod $num$;
          fi
          Xmit <$m.type, m.seqno, m.waitfor, m.data$> to $P_j$;
          if ($m.type =$ BF $\vee$ $m.type =$ 2F) then
            bfp($F$) := seqno($F$);
          fi
          sbuffer($F$) := sbuffer($F$) $\cup \{m\}$;
        fi
      else
        if |sbuffer($F$)| = 0 then
          Xmit <2F, $-1, -1, \Phi$> to $P_j$;
          sbuffer($F$) := sbuffer($F$) $\cup$ {<2F, $-1, -1, \Phi$>};
        fi
      fi
    fi
  od
end FS-Daemon
```

To deduce whether a newly arrived message is available for delivery in the bounded protocol, the receiver maintains slight variations of the sets **delv**(F) and **rbuffer**(F), and a new integer, **2Fdelv**(F). **delv**(F) is a set containing the sequence numbers of all messages which have been delivered at site P_j, *since* the delivery of the last two-way flush message. The set initially contains the element 0 (representing the implicit initial two-way flush transmission), the maximum number of elements is *num*, and the set is reset to one element whenever a two-way flush is delivered (actual or *dummy*). **rbuffer**(F), consistent with the original implementation, begins as an empty set. As mentioned, the size of this buffer is bounded by k. **2Fdelv**(F) is the sequence number representing the last two-way flush message that has been delivered. Initially, **2Fdelv**(F) is 0.

The daemon that implements the bounded receiving F-channel software requires minimal changes. In fact, the only changes concern the explicit command to ACK a message and the receipt of the *dummy* two-way flush.

```
Process FR–Daemon($P_i$ : site, $F$: FchannelID)
  while true do
    Recv <$m.type$, $m.seqno$, $m.waitfor$, $m.data$> from $P_i$;
    rbuffer($F$) := rbuffer($F$) ∪ {<$m.type$, $m.seqno$, $m.waitfor$, $m.data$>};
    change := true;
    if $m.seqno = -1$ then
      rbuffer($F$) := rbuffer($F$) − {$m$};
      delv($F$) := {0};
      2Fdelv($F$) := 0;
      change := false;
      Xmit <Ord, $m.seqno, -1, \Phi$> to $P_i$;
    fi
    while change do
      change := false;
      foreach $\mu \in$ rbuffer($F$) do
        BDealWith($\mu$);
        if $\mu.seqno \in$ delv($F$) then
          change := true;
          Xmit <Ord, $\mu.seqno, -1, \Phi$> to $P_i$;
        fi
      od
    od
    od
  end FR–Daemon
```

The main modification to DealWith, shown below as BDealWith, is the condition to determine the eligibility to deliver a two-way flush or forward flush message. Instead of verifying all messages previously transmitted have been delivered, BDealWith ensures all messages since the delivery of the last two-way flush (actual or *dummy*) have been delivered. Once the condition succeeds, the two-way flush or forward flush message is delivered. If the message is a two-way flush, then **2Fdelv**(F) is updated.

```
Procedure BDealWith($m$ : message)
  if ($m.type$ = Ord $\vee$ $m.type$ = BF) then
    if ($m.waitfor \in$ delv($F$)) then
      rbuffer($F$) := rbuffer($F$) $-$ $\{m\}$;
      delv($F$) := delv($F$) $\cup$ $\{m.seqno\}$;
      Deposit (<$m.type$, $m.data$>, Pipe($F$));
    fi
  else
    if (($\forall b : 0 \le b \le ((m.waitfor -$ 2Fdelv($F$) $+ num)$ mod $num)$) :
                        ((2Fdelv($F$) $+ b$) mod $num$) $\in$ delv($F$)) then
      rbuffer($F$) := rbuffer($F$) $-$ $\{m\}$;
      if ($m.type$ = 2F) then
        delv($F$) := $\{m.seqno\}$;
        2Fdelv($F$) := $m.seqno$;
      else
        delv($F$) := delv($F$) $\cup$ $\{m.seqno\}$;
      fi
      Deposit (<$m.type$, $m.data$>, Pipe($F$));
    fi
  fi
end BDealWith
```

2.7 Correctness of the Bounded WaitFor Technique

In order to prove the correctness of the bounded WaitFor technique, we find it convenient to define a new relation $\sqsubset_F$. If x and y are two messages transmitted over F-channel F, then $x \sqsubset_F y$ if and only if x is transmitted before y. Clearly, $\sqsubset_F$ is a total order, the transmission order of messages on F.

Lemma 7 *For a two-way flush or a forward flush message, m, in the bounded WaitFor protocol,*

$$Pred(m) = Pred(prev\mathbf{2F}) \cup \{prev\mathbf{2F}\} \cup \{m' : prev\mathbf{2F} \sqsubset_F m'\}.$$

Proof: By definition of a two-way flush or forward flush message, m, the predecessor set of m contains exactly those messages which were transmitted before m. That is,

$$Pred(m) = \{m' : m' \sqsubset_F m\}.$$

Since the bounded WaitFor protocol implicitly transmits a two-way flush message as the first message of each application, $prev\mathbf{2F}$ will never be empty. If $prev\mathbf{2F}$ is the implicit two-way flush message, then $Pred(prev\mathbf{2F})$ is empty. Otherwise,

$$Pred(prev\mathbf{2F}) = \{m' : m' \sqsubset_F prev\mathbf{2F}\}.$$

In either case,

$$Pred(m) = Pred(prev\mathbf{2F}) \cup \{prev\mathbf{2F}\} \cup \{m' : prev\mathbf{2F} \sqsubset_F m'\}$$ ∎

For an ordinary message or backward flush m, recall the *BFP-chain* of m as the set of messages

$$\text{chain}(m) = \{m_k, m_{k-1}, \ldots, m_1\}.$$

The BFP-chain is the path of backward flushes back to the closest two-way flush, including that two-way flush. In the bounded WaitFor protocol, m_k is guaranteed to be a two-way flush message due to the implicit transmission of the initial two-way flush message. We may now proceed with the safety theorem for the bounded WaitFor technique. This proof uses Lemma 3 from the correctness proof of the unbounded WaitFor protocol; however, the lemma continues to be valid.

Theorem 8 (SAFETY) *Under the bounded WaitFor technique, message m is consumed by a* **receive** *at the destination process only if Pred(m) has already been consumed.*

Proof: Consider the delivery of message m, a two-way flush or a forward flush message. The condition to deliver m in BDealWith is

$$\forall b : 0 \le b \le ((m.waitfor - \mathbf{2Fdelv}(F) + num) \textbf{ mod } num) :$$
$$((\mathbf{2Fdelv}(F) + b) \textbf{ mod } num) \in \mathbf{delv}(F)$$

By definition, num is the sequence number modulus. **2Fdelv**(F) is the sequence number representing the last two-way flush message that has been delivered. **delv**(F) is a set containing the sequence numbers of all messages which have been delivered, since the delivery of the last two-way flush message. $m.waitfor$ is one less (modulo num) than the sequence number of m. Therefore, the condition states that m is **Deposited** in **Pipe**(F) if all messages transmitted between the previous two-way flush (inclusive) and m have been **Deposited**. By Lemma 7, *Pred*(m) includes these messages and the predecessor set of the previously transmitted two-way flush message. To verify *Pred*($prev$**2F**) is **Deposited** before m, we refer to a previous lemma. The proof here is analogous to the proof in Lemma 5 and, therefore, omitted.

For an incoming backward flush or ordinary message m, the condition to deliver m in the bounded WaitFor technique is based on the delivery of $m.waitfor$. In FS–Daemon, $m.waitfor$ is the sequence number of m's backward flush point. In the definition of chain(m), m_1 is

m's backward flush point. BDealWith will not allow the delivery of m until after the delivery of m_1. Generalizing this argument to each backward flush on the BFP-chain of m, BDealWith will insist the **Deposits** are correctly ordered. The correctly ordered **Deposit** of *Head*(chain(m)) and of *Pred*(*Head*(chain(m))) are handled by the protocol for two-way flushes, which was shown to be correct in the first part of this proof. We therefore conclude, by Lemma 3, that *Pred*(m) must have been delivered before m is delivered.

As a final part of this argument, **Pipe**(F), in the bounded WaitFor technique, must be a FIFO buffer. Thus, **receives** will **Consume** messages in the same order in which BDealWith recognizes they are eligible for delivery and **Deposits** then in **Pipe**(F). ∎

Theorem 9 (LIVENESS) *Assume that the receiver will, in fact, issue a* **receive** *for each* **send** *executed by the sender. Message m, sent on the bounded WaitFor protocol of an F-channel, will be received in finite time.*

Proof: For the first part of the proof, let us assume that each message transmitted by a **send** operation is placed on the network. Since the system is assumed to be reliable, each message will arrive at the destination within finite time from its transmission and, therefore, the FR-Daemon will **Recv** the message in finite time.

We proceed by induction on the number of messages transmitted in the system. The basis case is transmitting m, the first message. This message, for any message type, will have *m.waitfor* set to 0. Since **delv**(F) initially contains 0, either of the delivery conditions will be satisfied in BDealWith and, therefore, m will be **Deposited** in **Pipe**(F) without delay. A **receive** will then **Consume** m within finite time.

Assume that the first n messages transmitted in the system are **Consumed** within finite time. Consider the case of m, the $(n + 1)st$ message transmitted in the system. By Theorem 8, m will not be **Deposited** in **Pipe**(F) until its predecessor set has been **Deposited**. We need to show, however, that the receiver will maintain information so m can verify its predecessor set has been **Deposited**.

Suppose m is a two-way flush or forward flush message. *m.waitfor* will then be one less than *m.seqno* (modulo *num*). The delivery condition for m requires all messages transmitted since the last two-way flush, including that two-way flush, be a member of **delv**(F). By the inductive hypothesis, we know these messages will be **Consumed** in finite time. Without regards to re-initialization, **delv**(F) maintains the sequence numbers of all messages delivered *since* the delivery of the last two-way flush. We, therefore, conclude m will be **Deposited** and **Consumed** in finite time as well.

Suppose m is a backward flush or ordinary message. Recall that *m.waitfor* is the sequence number of the backward flush point at the transmission of m. Since the backward flush point will be a message in the first n messages transmitted, we know, by the inductive hypothesis, it will be **Consumed** within finite time. The set **delv**(F) contains the sequence numbers of all messages that have been delivered *since* the delivery of the last two-way flush message. Furthermore, Lemma 2 proved no message will be "between" a backward flush point and a backward flush or ordinary message. Therefore, *m.waitfor*, referring to a two-way flush or backward flush message, will be an element of **delv**(F) in finite time and will not have the possibility of removal until after the delivery of m.

As a final part of this argument, it is essential that **delv**(F) is re-initialized only when the system is empty of messages. The sender transmits a *dummy* two-way flush message only when the system is empty and the receiver re-initializes its variables only when the *dummy* two-way flush is delivered. Hence, **delv**(F) will contain the messages required to satisfy the delivery condition for each transmitted message in finite time.

In the first part of the proof, we assumed each message transmitted by a **send** operation is placed on the network. At the sender of the bounded WaitFor technique, however, a message will not be placed on the channel by a **Xmit** operation unless the transmission condition is verified:

$$|\mathbf{sbuffer}(F)| < k \wedge (\mathbf{seqno}(F) + 1) \ \mathbf{mod}\ num \neq \mathbf{2Fack}(F).$$

We now show that this condition, when false, will become true in finite time. In the first part of the proof, we verified that if a message is transmitted on the channel, then it will be received in finite time. At each delivery of a message m, the receiver transmits an ACK for m to the sender. The sender will receive an ACK for each message it transmitted in finite time and will thus remove messages from **sbuffer**(F) in finite time. For the transmission condition to be made true in finite time, we must also show that **2Fack**(F) is modified in finite time. If a two-way flush message is transmitted, **2Fack**(F) is updated. If, however, no two-way flush message is transmitted in *num* messages, then the system will empty itself of all messages and the transmission condition will continue to be false. In this scenario, the sender transmits a *dummy* two-way flush, thus re-initializing all the variables. Since we assume $num > 1$ and the initial values of **seqno**(F) and **2Fack**(F) are both 0, the transmission condition will become verified and each message from a **send** operation will be placed on the network by a **Xmit** operation in finite time.

Lastly, both sides of the F-channel exhibit liveness as long as messages are initially placed on the network. That is, in the receiver's argument, we assumed each message to be transmitted is placed on the network. In the sender's proof, we based the argument on the assumption that the sender receives ACKs for previously transmitted messages. Since the transmission condition will not delay the transmission of the first k messages, the sender will begin placing messages on the network and the receiver will transmit ACKs for these messages in finite time. Therefore, messages sent on the bounded WaitFor technique will be transmitted and received in finite time. ∎

2.8 Performance Considerations

2.8.1 The System Model

An F-channel offers the implementor of a distributed application the flexibility of specifying a message delivery order apropos of the demands of the application. This stands in marked contrast to the rigid FIFO delivery order imposed upon the application by virtual circuit communication. Intuitively, the more restrictive the delivery order, the less concurrency available to exploit in message transmission over multiple links. In the next two sections, we investigate the gain in effective network bandwidth when ordinary messages are batched together by a flush message and are transmitted over a multi-link F-channel, as compared to messages transmitted over a multi-link virtual circuit. We assume the implementation of the F-channel is similar in spirit to that of the WaitFor technique; that is, one copy of each message is transmitted across the F-channel. First, simulation results in Section 2.9 show that the relaxed delivery order restrictions of the F-channel may reduce the mean delivery time of a batch by a factor of three or four—this difference may be critical in meeting real-time requirements of the application. Second, in Section 2.10, analytical results are derived to validate the simulation results.

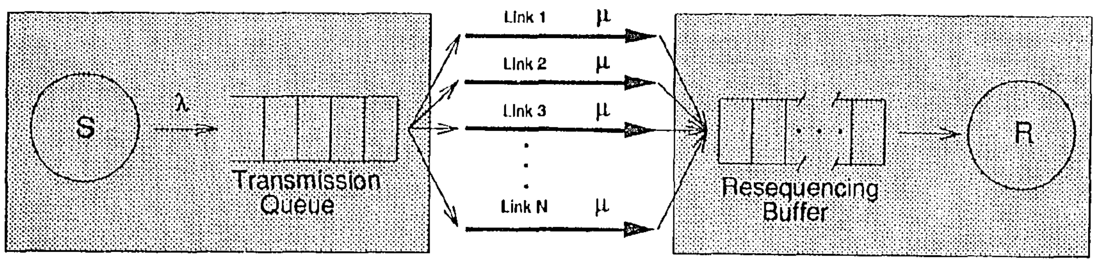

Figure 2.9: Performance System Model

Consider the system model shown in Figure 2.9. Regardless of the communication paradigm, the figure illustrates two communicating processes connected by N separate and independent links. Messages to be transmitted are generated by the sender (denoted by S).

Upon generation, if all N links are busy, the message is placed in the FIFO transmission queue. If, on the other hand, a transmission link is available, the message is immediately transmitted. In this model, we assume that the transmission queue is unbounded. We also assume the existence of an underlying network layer mechanism which assures reliable transmission of messages.

As stated, messages may arrive at the receiver (denoted by R) in an unpredictable order. The ultimate delivery of a message, as discussed in the beginning of this chapter, may be delayed due to two reasons. First, for any communication paradigm (FIFO, unordered, and flush), the receiver may not have issued a receive command. Second, for both FIFO and flush communication paradigms, a message cannot be delivered until all ordering restrictions are satisfied, i.e., the resequencing delay. This second delay occurs while the message waits in the resequencing buffer. As we did for the transmission queue, we assume the resequencing buffer is unbounded in size.

2.8.2 Data Batched by Flush Messages

We have found, for every F-channel application developed to date, real-world examples naturally form *batches* of ordinary messages and an associated flush message of a given type [Ahu90, AVS91, CK91, CKA93]. Any flush application that transmits information from the sender to the receiver and uses more than one flush type appears to be a contrived example. Although each flush type is independently beneficial, we question the usefulness of transmitting more than one flush type on an F-channel. Therefore, in the performance sections, we consider a message passing scenario partitioned into batches of ordinary messages and a singular flush message.

Following a batch of ordinary messages with a forward flush effectively "closes" the batch. The ordinary messages (from all batches) may be delivered in any order, but all ordinary messages in one batch must be delivered before the batch-terminating forward flush is delivered. Batches of ordinary messages separated by a two-way flush completely isolates batches from one another; i.e., all messages in batch i will be delivered before any message in batch $i+1$ is delivered. Using a backward flush to precede a batch of ordinary messages "announces" the coming batch. No ordinary message in a given batch may be delivered until its preceding backward flush, but given that restriction, the delivery of all ordinary messages (for all batches) is unordered. In the following discussion, we consider the three batching scenarios in more detail. In each case, we suppose a batch, consisting of B messages, represents a single frame in the transmission of digital image information. Each ordinary message in a given batch contains image data for a small region of the display area and the identity of the region in which it should be displayed. The receiver constructs the frame in pieces—as ordinary messages arrive, its sub-image is pasted into the appropriate position. Each message consists of three fields. The first field is the type of the message; the second field indicates the batch number to which the message belongs; the third field, for an ordinary message, denotes the number of the message within the batch; the third field in a flush message is zero.

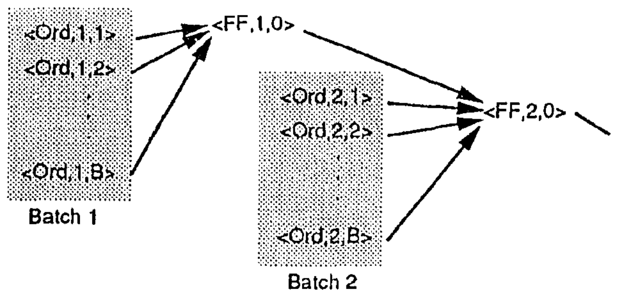

Figure 2.10: Batches Terminated with Forward Flushes

Let us again consider batching ordinary messages with forward flush messages. Figure 2.10 shows the immediate predecessor graph for this application. We see that delivery of *all* of the ordinary messages is unordered (with respect to other ordinary messages). The destination process expects, however, all of the ordinary messages in a batch to be delivered before the forward flush "closes" the batch. The delivery of a forward flush signals that its entire frame has been delivered. One application of this batching protocol could be the storage of the image frames in separate files. The receipt of a forward flush signals that the entire frame has been received and stored, and that the file in which it has been stored may be closed. As an end-to-end image integrity convention, the terminating forward flush could contain a checksum; the destination process computes the checksum function incrementally as the ordinary messages stream in. When the forward flush arrives, the checksum in the message is compared to the receiver's computed checksum before the frame is finally accepted.

If we let two-way flush messages delimit the ordinary messages, we see a different partial order—Figure 2.11 illustrates this batching scenario. The ordinary messages which constitute a batch may arrive in any order, but a two-way flush ensures that all messages in one batch are delivered before any message in the next batch. Suppose we wish to transmit a group of images from one site to another for real-time animation. The individual frames are constructed in the display buffer as the ordinary messages arrive at the destination. Upon receipt of a two-way flush, the contents of the buffer are physically displayed, replacing the previously displayed frame.

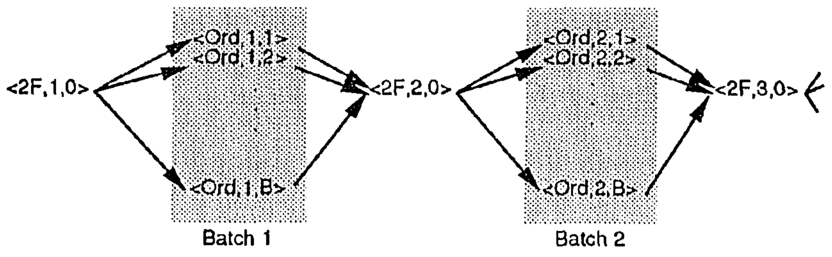

Figure 2.11: Batches Separated with Two-way Flush Messages

Batching ordinary messages with backward flush messages can also be applied to the transmission of digital image information. Figure 2.12 illustrates this message passing scenario. In this situation, the backward flush effectively "announces" the coming batch, presumably providing the consumer process with information used to define the ordinary messages included in the batch. As in the forward flush case, we see that *all* of the ordinary

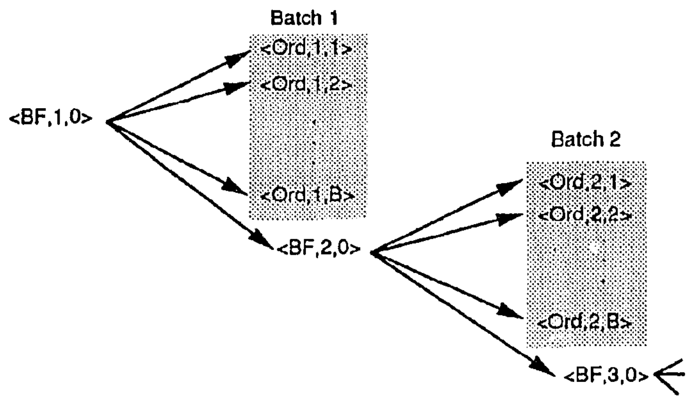

Figure 2.12: Batches Preceded by Backward Flushes

messages may be delivered in any order, but that each ordinary message must be delivered after the backward flush announcing the batch to which it belongs. Again suppose we wish to transmit a group of images from one site to another. Further suppose the display memory of the consumer is a scarce resource and that the backward flush includes the size of the image conveyed in its batch. The consumer, upon receipt of a backward flush, may calculate if the image will fit on the display. If so, the appropriate amount of display memory is reserved, and the incoming image is displayed as it is received. If the image is too large, then it is stored in a file for later display.

2.9 Simulation Results

Once again, consider the system model shown in Figure 2.9. In the simulation results presented here, we compare the transmission of the batched data examples on an F-channel with a virtual circuit over N transmission paths. The message generation process is assumed to be Poisson; thus, the message inter-generation interval is exponential with mean $1/\lambda$. The transmission time on any link is, initially, an exponential random variable with mean $1/\mu$. Later in this section, we replace the exponential random variable with a hyperexponential random variable and consider the effect on the system when the variance in transmission times is increased. Initially, however, we have an exponential random variable with the value of μ fixed at one; N, λ, and B are experimental parameters. (Recall that B is the number of ordinary messages in a batch.) We define ρ, the system utilization, as

$$\begin{aligned}\rho &= \lambda/N\mu, \\ &= \lambda/N \qquad \text{when } \mu = 1.0.\end{aligned}$$

We may model this system, excluding the resequencing buffer, as an $M/M/N$ queue in which the condition for reaching steady state is that $\rho = \frac{\lambda}{N} < 1$. We insist this equilibrium condition holds in all the simulation trials. Since we do not know the regeneration points in this system, we use the method of batched means to estimate steady state. An interval estimate for the unknown mean is then calculated on the means from the batched data with 95% confidence. Each run in the simulation consists of processing 200,000 messages; each batch size is 5,000 and, therefore, the number of batches in each run is 40.

We use mean message delay, $\bar{D}$, as the principal performance metric. Mean message delay includes queueing delay at the transmitter, time on the physical network link, and resequencing delay at the destination process; it is the mean end-to-end message transmission time (excluding any delay due to the lack of a **receive** at the destination process). $\bar{D}$ thus indicates the mean delay from time of arrival until a message is made eligible for delivery.

2.9.1 Multi-link Virtual Circuit

As a benchmark, we consider the performance of a virtual circuit in Figures 2.13 and 2.14. The analysis of resequencing delay in [AR87] validates these two figures. Figure 2.13 illustrates the three individual components that complete the mean message delay for a virtual circuit implemented on 25 transmission links. We plot utilization, ρ, versus delay, showing 95% confidence intervals. The figure exhibits that resequencing delay is an important factor in the total delay. In fact, resequencing delay dominates transmission delay and queueing

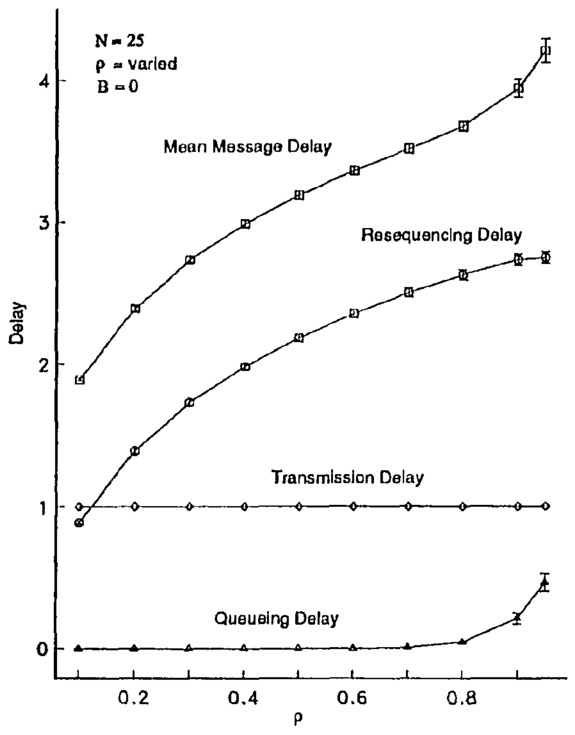

Figure 2.13: Message Delays on a Virtual Circuit

delay for almost all utilizations. Although the exact form of the plot depends substantially on the fact that there are 25 links, any system with more than one link reveals that resequencing delay is an important factor in the total delay.

Unlike Figure 2.13, we do not plot a confidence interval in the subsequent figures of this thesis. Our simulator, however, calculated every data point with a 95% confidence interval. In each simulation trial, the confidence intervals gradually increase as utilization increases. Since none of the confidence intervals are very large (in fact, most are drawn smaller than the symbol used to represent the mean on our plots), we omit them.

In Figure 2.14, we plot utilization, ρ, versus mean message delay, $\bar{D}$. In this plot, we consider the effect of varying the number of links on which the virtual circuit is implemented. At utilizations less than 0.5, we find that $\bar{D}$ is exactly *opposite* what one would expect. Adding links to the system at low utilization *increases* the delay. In fact, a 100-link virtual circuit has a mean message delay *higher* than a single link virtual circuit until the utilization approaches 0.8. This result is explained by the importance of resequencing delay as links are added to the system. With fewer links, queueing delay becomes more important at higher utilizations. We see, for example, that a 25-link virtual circuit becomes faster than an 8-link virtual circuit when utilization is greater than 0.9. A virtual circuit's insistence on a FIFO delivery order leads to a mean message delay that is non-monotonic (with respect to N) and counterintuitive. In summary, increasing the number of physical links between two processes communicating in a FIFO manner does not necessarily result in a higher effective bandwidth.

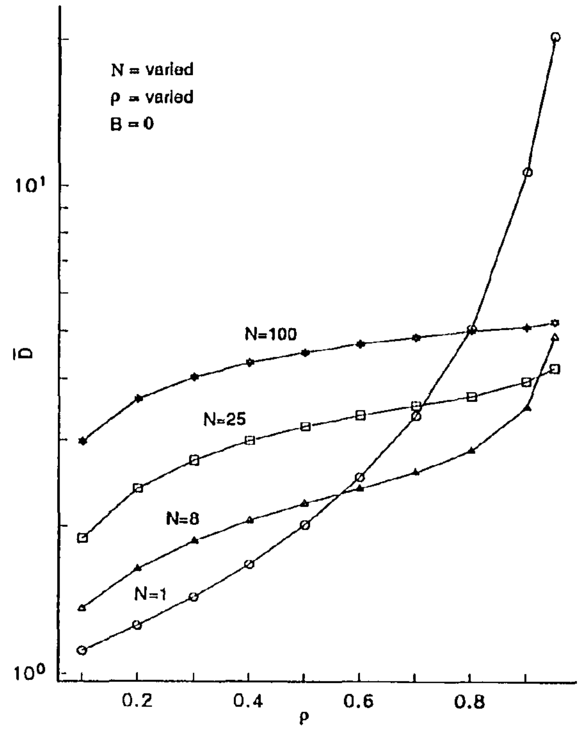

Figure 2.14: Virtual Circuit Mean Message Delay

As a comparison, we see monotonic, predictable behavior in Figure 2.15. In this plot, we have, basically, no delivery order restrictions—a single batch is transmitted (199,999 ordinary messages followed by a forward flush). In effect, the figure illustrates the mean message delay for reliable datagram communication. As in Figure 2.14, the vertical axis is mean message delay, and the horizontal axis is utilization. We immediately notice that the system *always* benefits (decreases $\bar{D}$) from the addition of communication links. Furthermore, every mean message delay in Figure 2.15 is less than the corresponding mean in Figure 2.14. The mean message delays are equivalent, however, when a single communication link connects the two processes, We realize the difference in the two plots is completely due to the number of restrictions placed on the delivery order, i.e., a *degree* of order. F-channels allow the user to specify these two example degrees of order. In addition, F-channels allow many degrees of order between these two extremes. The following section investigates the impact of degree of order upon mean message delay.

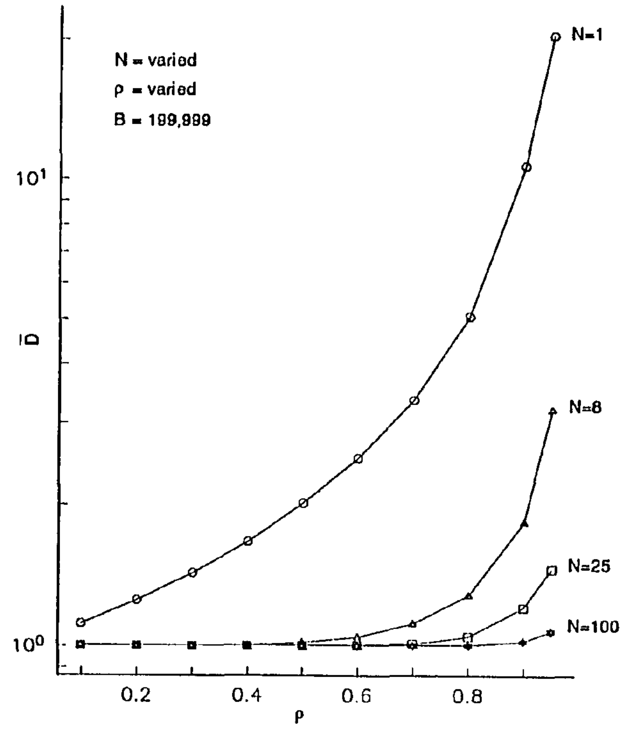

Figure 2.15: Reliable Datagram Mean Message Delay

2.9.2 The Effect of Delivery Order Restrictions

In this section, we investigate the effect of delivery order restrictions on the three different batching scenarios. All three plots in this section, Figures 2.16 to 2.18, keep the number of communication links between the sender and receiver fixed at 25 and we plot mean message delay versus the number of batches transmitted. The fewer the number of batches means the less the degree of order. The less order means the greater the potential that a multi-link channel can exploit concurrent message transmission without incurring excessive resequencing delays. At the left end of the horizontal axis there are, basically, no message delivery restrictions; a single batch is transmitted. In effect, we have reliable datagrams. At the right end of the horizontal axis, the mean message delay is equivalent to the mean message delay if the messages were transmitted across a virtual circuit—200,000 flush messages are processed. In the middle of the horizontal axis, for instance, we transmit 100 batches of 1,999 ordinary messages (delimited by a flush message). Clearly, the horizontal

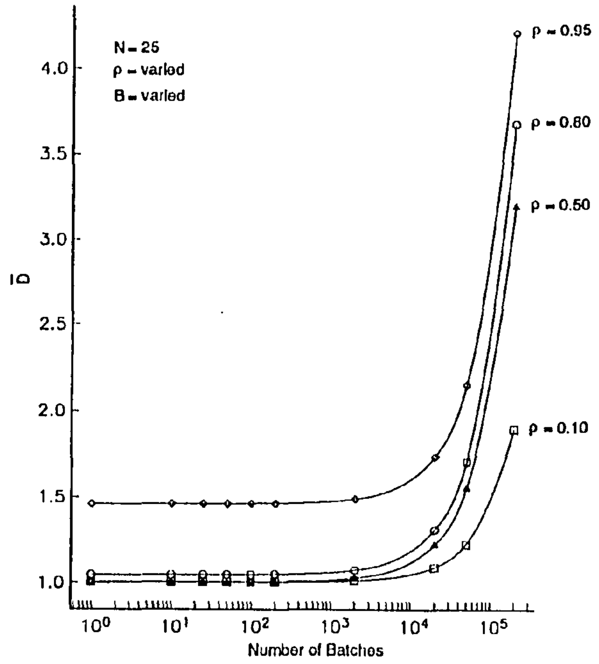

Figure 2.16: Forward Flush Batching Scenario

axis represents all possible degrees of order.

In the first of these three plots, Figure 2.16, we consider the impact of delivery order restrictions when transmitting batches of ordinary messages terminated by forward flush messages. The plot illustrates that mean message delay monotonically increases as delivery order restrictions increase. Furthermore, there is a tremendous increase in $\bar{D}$ as the degree of order goes from 20,000 to 200,000 batches for *every* utilization.

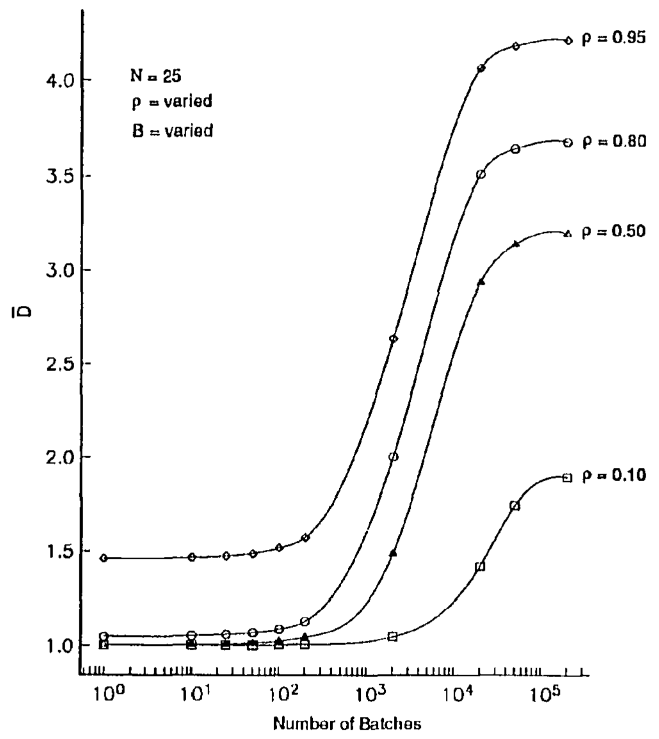

Figure 2.17: Two-way Flush Batching Scenario

In Figure 2.17, we consider the degree of order impact when we transmit batches of ordinary messages delimited by two-way flush messages. In this case, the dramatic increase in $\bar{D}$ begins when the number of batches transmitted is 200. Consider a utilization of 0.5. When the number of batches increases from 200 to 20,000, the mean message delay increases by almost 300%.

Now let us compare batching ordinary messages by backward flush messages (Fig-

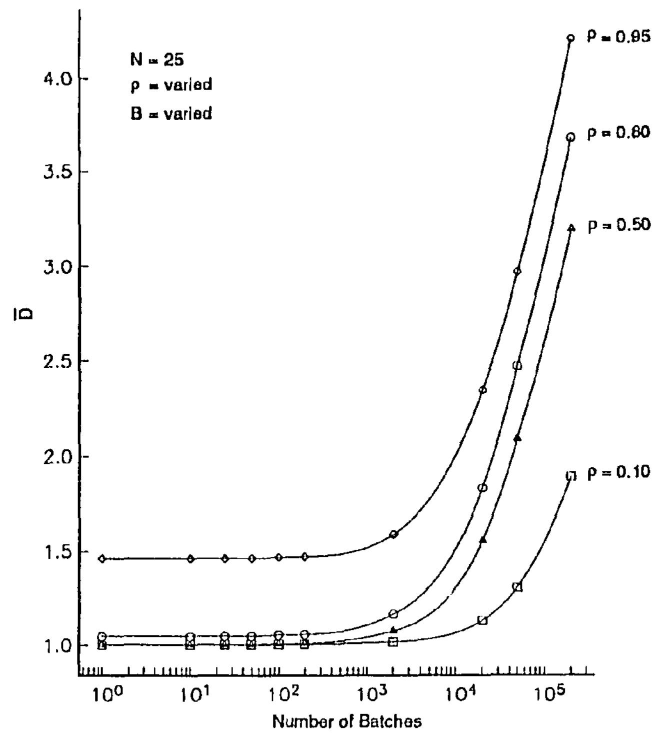

Figure 2.18: Backward Flush Batching Scenario

ure 2.18) with the two-way flush batching scenario. Again we see $\bar{D}$ monotonically increasing as delivery order restrictions increase. Similar to the two-way flush batching scenario, there is a dramatic increase in the mean message delay. In this case, however, the increase does not begin until the number of batches transmitted becomes 2,000. Looking back at Figure 2.17, we find the large increase in the mean message delay begins when the number of batches is 200. In addition, $\bar{D}$ for the backward flush batching scenario is almost consistently less than $\bar{D}$ for the two-way flush batching scenario. The only exceptions are at the two extreme ends of the degree of order: (basically) no order and total order. Take, for example, transmitting 2,000 batches, of batch size 99, when utilization is 0.8. The delivery of these batches will take 70% longer if the batches are delimited with two-way flush messages instead of backward flush messages. In summary, batching ordinary messages with backward flush messages has lower mean message delays than batching ordinary messages with two-way flush messages.

As a final comparison between Figures 2.16 to 2.18, consider transmitting 20,000 batches, of size nine, when utilization is 0.8.

Batching Scenario	Mean Message Delay
FF	1.31
BF	1.83
2F	3.50
VC	3.67

In this situation, it takes 40% longer to use backward flush batching, 160% longer to use two-way flush batching, and 180% longer to use virtual circuit communication instead of transmitting the batches with forward flush messages. In this section, we throughly examined the impact of degree of order upon mean message delay for each batching scenario. For each batching scenario and each utilization, the mean message delay reaches the mean message delay of virtual circuit communication in the worst possible case only. In the following section, we plot the impact of degree of order, given utilization of 0.5, for each batching scenario in one concise plot. In addition, the section analyzes the effect of our other two experimental parameters: ρ and N.

2.9.3 The Effect of B, ρ, and N

In the simulation results presented in this section, we compare the three batching scenarios and virtual circuit communication. In each of the three plots, we keep two of the experimental parameters fixed and examine the effect of mean message delay when the third parameter is varied. Figure 2.19 brings together the batching scenario results of the previous section for utilization of 0.5. As illustrated, the mean message delay, for any given F-channel batching possibility, monotonically increases as the amount of delivery restric-

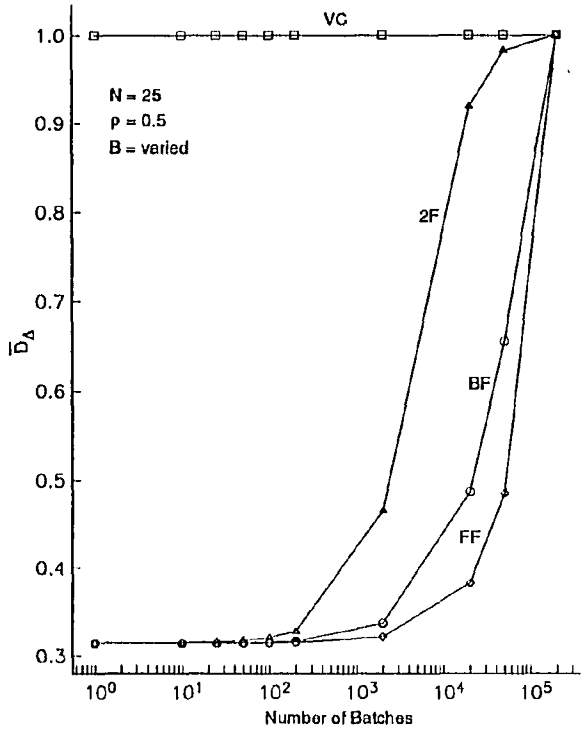

Figure 2.19: Varying the Number of Batches

tions placed on message delivery increases. The figure considers a 25-link system and a utilization of 0.5. In this plot, each point represents the difference between the mean message delay for a virtual circuit and the mean message delay for a given communication scenario when the number of batches is varied. (A given communication scenario is either virtual circuit communication or F-channel communication utilizing a named flush type to delimit the batches.) Each point in the plot is computed as

$$\bar{D}_\Delta = \frac{\bar{D}_X}{\bar{D}_{VC}}$$

where the subscript of $\bar{D}$ represents the communication scenario. A value of $\bar{D}_\Delta$ close to 1 indicates minimal gains for batched data delivery over a multi-link F-channel compared to a virtual circuit over the same number of links. The smaller the value of $\bar{D}_\Delta$, the greater the performance improvement. The value of $\bar{D}_\Delta$ is simply the fraction of the virtual circuit delay produced by the F-channel batching paradigm. For example, a value of 0.4 for $\bar{D}_\Delta$ means the specified batching scenario takes only 40% the time of transmitting these messages on a virtual circuit. We immediately note the performance gain a user will obtain if messages can be transmitted using an F-channel. In the best situation, message delay for a batching application communicating with an F-channel will be approximately one third that of the same application communicating with a virtual circuit. The delay for an F-channel approaches that of a virtual circuit in the worst case only. If the semantics of the application permit, F-channels offer promise of providing high bandwidth inter-process communication. Since the flush communication paradigm allows the user to specify the least delivery restrictions necessary for the application, the best mean message delay can be obtained.

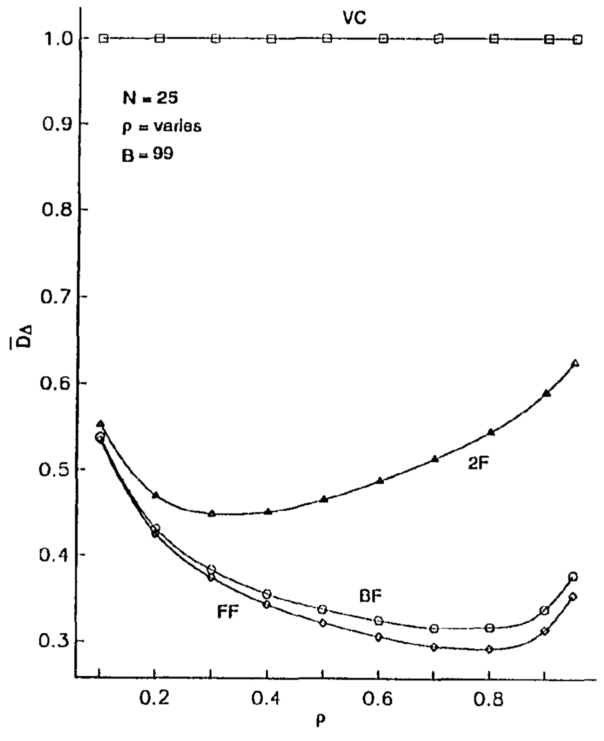

Figure 2.20: Varying the System Utilization

Since we have considered the degree of order effect on each batching scenario, we now analyze the effect of our other two experimental parameters: utilization and number of links. In Figure 2.20, we consider utilization. As in the previous figure, we plot $\bar{D}_\Delta$ on the vertical axis. In this case, however, we fix the batch size at 99 and ρ is the independent variable. The plot illustrates that both forward flush and backward flush batching scenarios have the best gain in performance (compared to virtual circuit communication) when utilization is 0.8. Ordinary messages separated by two-way flush messages, on the other hand, are best at about a 0.3 utilization. As utilization increases after the best gain in performance for each batching scenario, we realize that the queueing delay begins to override the resequencing delay benefits. In each batching scenario, however, we easily see the benefits of communicating with an F-channel.

Figure 2.21 considers the effect of varying the number of links. In this simulation experi-

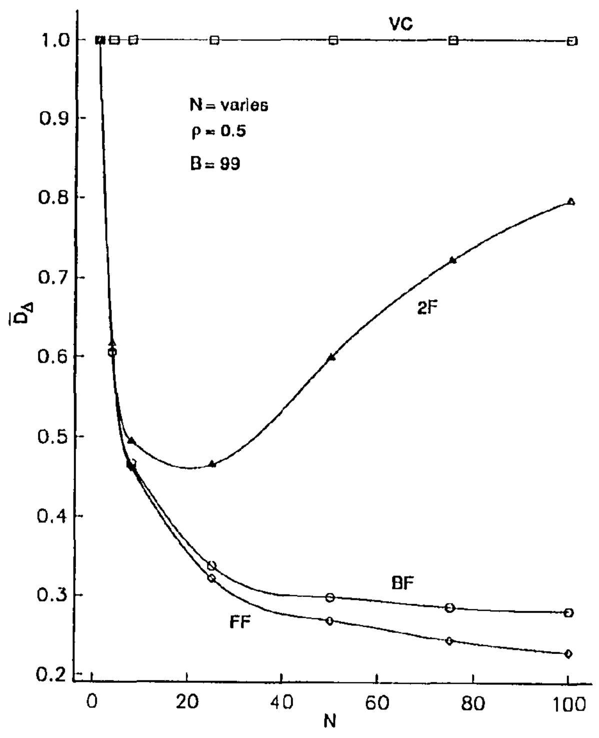

Figure 2.21: Varying the Number of Links

ment, we fix the batch size at 99 and utilization at 0.5. The two-way flush batching scenario exhibits its best performance when the number of links in the system is approximately 20. As in the virtual circuit communication case, the mean message delay for batching ordinary messages with two-way flush messages degrades when more than 20 links exist in the system. (This result is heavily dependent on the fixed parameters.) The backward flush and forward flush batching scenarios, on the other hand, do not show a degradation even as the number of links in the system increases to 100. When we consider a 100-link system and batch ordinary messages with forward flush messages, we see the mean message delay is less than 25% of virtual circuit communication.

In this section, we compared the performance of virtual circuit communication with the three batching scenarios in F-channel communication. All the results illustrate that a programmer can obtain much faster data transmission if batches of ordinary messages delimited with a flush message of a given type are transmitted on an F-channel. In the three F-channel batching paradigms, there is a clear correspondence between the degree of disorder allowed in message delivery and the potential for effective concurrent message transmission without excessive resequencing delay. In other words, referring to Figures 2.10 to 2.12 of Section 2.8.2, the fact that batching with forward flush messages is less restrictive than batching with backward flush messages, which, in turn, is less restrictive than batching with two-way flush messages is reflected directly in the simulation result of Figure 2.19. Furthermore, in all three batching scenarios, the larger batch sizes have smaller mean resequencing delays leading to smaller mean delays. That is, as the degree of order increases, there is a monotonic increase in the mean message delay.

A second conclusion may be drawn from Figure 2.20. Resequencing delay is more important in mid-range utilizations and, hence, $\bar{D}_{\Delta}$ is lowest for utilizations between 30% and 70%. As discovered in virtual circuit communication and shown in Figure 2.13, resequencing delay is less of a factor in the total delay when utilization is low, due to less out-of-order message delivery, and when utilization is high, due to the increased importance of queueing delay. Figure 2.21 illustrates that the backward flush and forward flush batching scenarios exploit as many links as available. The more restrictive, in terms of message delivery order, two-way flush batching scenario has a response similar to virtual circuit communication; i.e., when the number of links in the system is increased, the mean delay is also increased. The performance of the two-way flush batching scenario of batch size 99 does improve, however, as the number of links in the system goes from 1 to 20. Having more than 20 links in the system and adding additional links decreases the performance gain in the two-way flush batching scenario.

It is also evident from the comparative plots of Figures 2.19–2.21 in this section that the performance of forward flush batching and backward flush batching are quite similar, but substantially better than the performance of two-way flush batching. Two-way flush batching, however, still outperforms a virtual circuit by a wide margin.

In this section, we analyzed the effect of varying our three experimental parameters on mean message delay. We note that in modeling the transmission time distribution as exponential with parameter μ, we have ignored the effects of the transmission time distribution on the message delay. It is known [Cho89] that the resequencing delay for multi-link virtual circuits is sensitive to higher moments of the transmission time distribution. As a trivial example, if message transmission time is fixed at the constant value $1/\mu$, there is no resequencing delay. All of our results have an implicit assumption that the sender and receiver are linked across a packet switched internetwork and, therefore, we expect message transmission time to be highly variable. In the following section, we examine the effect of the second moment of the transmission time.

2.9.4 The Effect of Variance

Although the results of this section continue to be simulation-based performance results, we separate the preceding section with the results presented here due to the extreme differences in the underlying system. In this section, the message generation process continues to be

Poisson with mean λ, but the transmission time on any link is a 2-stage hyperexponential random variable with mean $1/\mu$ and squared coefficient of variation C^2 where

$$C^2 = \frac{\text{variance}}{(1/\mu)^2}.$$

When a message is ready for transmission, it has probability α_1 to transmit on a link which is exponentially distributed with mean $1/\mu_1$ and it has probability $(1 - \alpha_1)$ to transmit on a link which is exponentially distributed with mean $1/\mu_2$. Therefore,

$$\frac{1}{\mu} = \frac{\alpha_1}{\mu_1} + \frac{1 - \alpha_1}{\mu_2}.$$

We may model this system, excluding the resequencing buffer, as an $M/H_2/N$ queue. In the following simulation results, μ continues to be fixed at one and C^2 becomes an experimental parameter. We plot C^2 versus $\bar{R}_\Delta$, the fraction of the virtual circuit *resequencing* delay produced by the F-channel batching paradigm:

$$\bar{R}_\Delta = \frac{\bar{R}_X}{\bar{R}_{VC}}$$

where X represents a particular batching scenario.

In Figures 2.22 to 2.24, we keep the number of communication links and utilization fixed at eight and 0.5 respectively. At the left end of the horizontal axis, where $C^2 = 1.0001$, $\bar{R}_\Delta$ represents the performance improvement of the particular batching scenario over virtual circuit communication when the message transmission distribution is approximately exponential. As C^2 increases, the three figures illustrate that the three batching scenarios produce different degrees of sensitivity to increasing the variation in transmission times.

In the first of the next three plots, Figure 2.22, we consider the effect of variance when transmitting batches of ordinary messages delimited by two-way flush messages. The plot illustrates that the two-way flush batching scenario is extremely sensitive to increases in

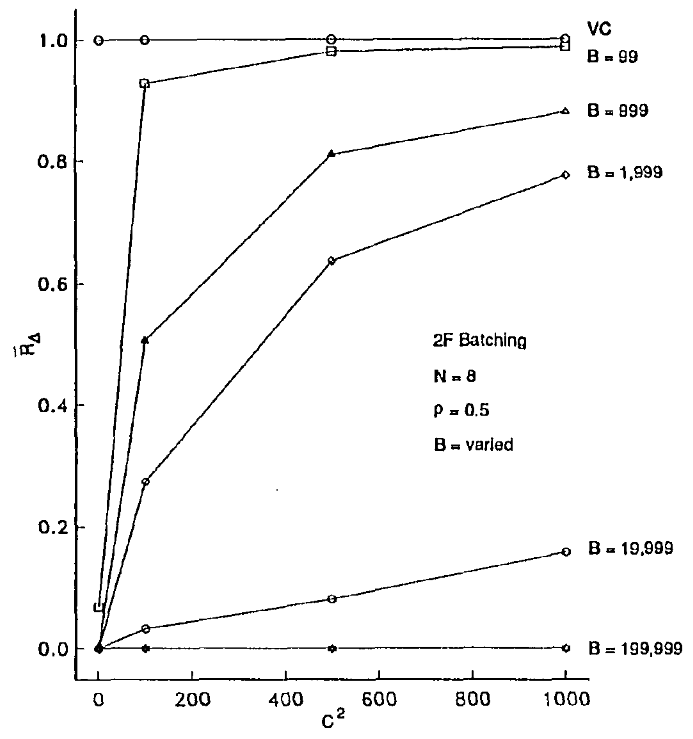

Figure 2.22: Hyperexponential: Varying the Number of Batches

the coefficient of variation. As C^2 increases, the performance gain of the two-way flush batching paradigm is decreased. A batch size of 199,999 (basically no resequencing delay) is the only batch size not affected by the hyperexponential distribution. Consider a batch size of 999. When C^2 increases from 1.0001 to 500, $\bar{R}_\Delta$ increases from 0.5% to 80%.

In Figure 2.23, we consider the effect of a 2-stage hyperexponential transmission distribution on the transmission of ordinary messages terminated by forward flush messages. In this case, the effect of variance is not as severe. In fact, the performance gain of each batch size over virtual circuit communication remains unchanged for every C^2. Consider a batch size of one. 50% of the messages have no resequencing delay and 50% of the messages have a resequencing delay as if the message were transmitted across a virtual circuit. Therefore, regardless of C^2, the mean resequencing delay of a message in this batching scenario is one half the mean resequencing delay of a message transmitted across a virtual circuit. Every message transmitted across a virtual circuit is sensitive to increases in C^2; in the forward flush batching paradigm, only forward flush messages are affected by increases in C^2.

Figure 2.24 illustrates the effect of variance when the ordinary messages are preceded by backward flush messages. As C^2 increases, $\bar{R}_\Delta$ *decreases*. Again consider a batch size of one. When $C^2 = 1.001$, $\bar{R}_\Delta = 64\%$. When $C^2 = 1000$, $\bar{R}_\Delta = 42\%$. Recall the comparative plots of Figures 2.19 to 2.21. Although the forward flush and backward flush batching scenario results are similar, the forward flush batching scenario continuously outperforms the backward flush batching scenario. This result is interchanged when we consider the effect of variance. As C^2 increases, the backward flush batching scenario begins to outperform the forward flush batching scenario.

Figures 2.25 and 2.26 analyze the effect of utilization and number of links on the two-way flush batching scenario, the F-channel batching paradigm that exhibits the least amount

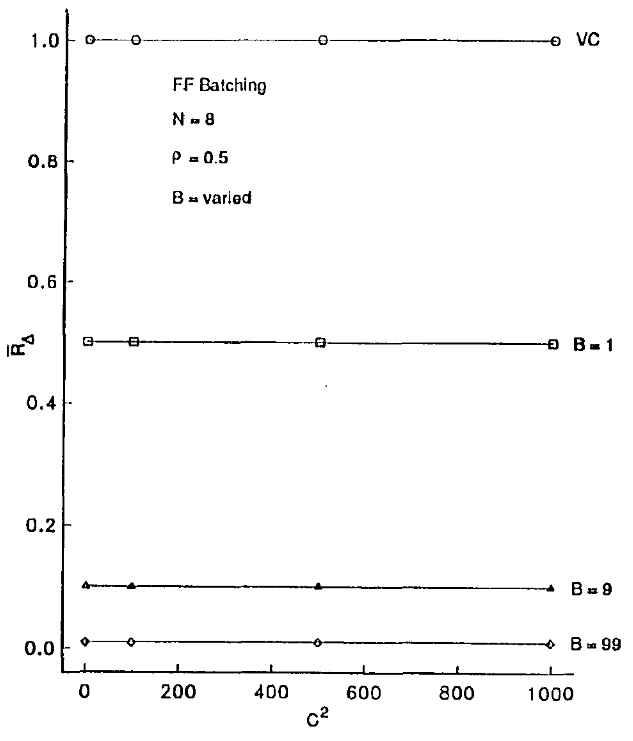

Figure 2.23: Hyperexponential: Varying the Number of Batches

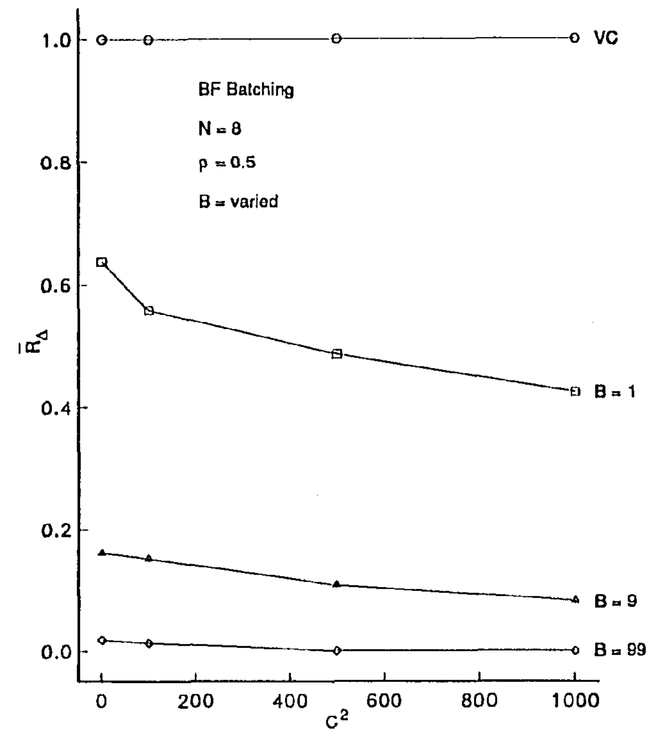

Figure 2.24: Hyperexponential: Varying the Number of Batches

of performance gain when the effect of variance is examined. In Figure 2.25, we consider utilization. As in the previous figures, we plot C^2 on the horizontal axis and $\bar{R}_\Delta$ on the vertical axis. In this case, however, we fix the batch size at 19,999 and consider the effect of varying ρ. Higher utilizations lose the performance gain faster than lower utilizations when C^2 is increased. This result is intuitive; higher utilizations will be more adversely affected by greater variation in the transmission times.

Figure 2.26 considers the effect of variance on the two-way flush batching scenario when we alter the number of links in the system. In this simulation result, we fix the batch size at 19,999 and utilization at 0.5. In all cases, the performance gain is reduced as C^2 is increased. Consider a 25-link system. When $C^2 = 100$, $\bar{R}_\Delta = 8\%$, but when $C^2 = 1000$, $\bar{R}_\Delta = 49\%$. As more links in the system allow more opportunity for messages to arrive out-of-order, the performance gain over a virtual circuit is reduced.

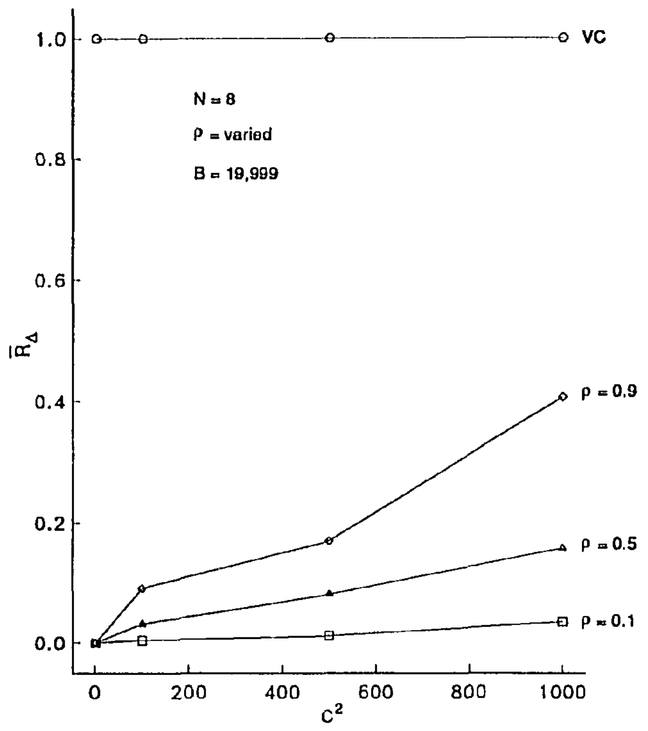

Figure 2.25: Hyperexponential: Varying the System Utilization

One way to comprehend the effect of an increase in the variation of message transmission, is to deduce the number of messages that are affected when the transmission time of one message is substantially increased. In the two-way flush batching scenario, one message can delay the delivery of all messages in subsequent batches. Therefore, this batching scenario is extremely sensitive to increases in C^2. In fact, batch sizes less than 99 and coefficient of variations greater than 10 exhibit almost no performance gain over transmitting the messages across a virtual circuit. As C^2 is increased in a system transmitting forward flush batches, the performance gain over a virtual circuit remains unchanged. In other words, one slow message can delay the delivery of all subsequently transmitted forward flush messages; however, the delivery of ordinary messages is unaffected by a message with a long transmission time. The backward flush batching scenario illustrates a performance improvement over virtual circuit communication when variance is considered. In this batching scenario,

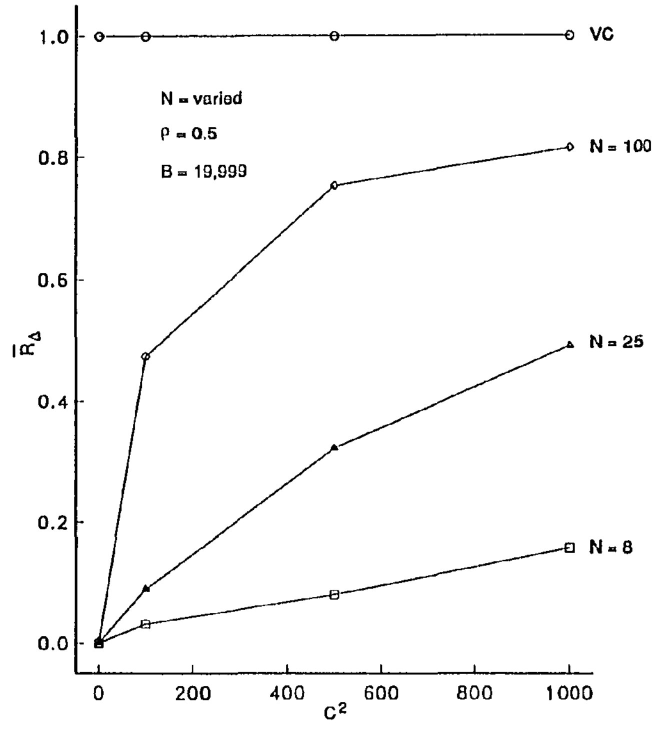

Figure 2.26: Hyperexponential: Varying the Number of Links

only the transmission of backward flush messages can delay the delivery of other messages in the system. Since fewer messages can affect the delivery of subsequently transmitted messages, the batching paradigm is less sensitive to increases in C^2.

For every batching scenario and every coefficient of variation, the resequencing delay of a message transmitted across an F-channel is as good as, if not better, than the resequencing delay of a message transmitted across a virtual circuit. While increasing C^2 does adversely affect the performance gain of a message in the two-way flush batching scenario, F-channel communication continues to outperform a virtual circuit.

We note that a central developmental trend in computation is that we (generally) achieve speed through the concurrent operation of many processors. The magnitude of such a speed-up depends upon the nature of the computation and how well it can be partitioned into concurrent activities. A similar phenomenon is obvious in this work—the use of F-channels provides the opportunity for concurrent message delivery over multiple links. F-channel communication can be used to relax delivery order restrictions and, hence, to make data transfer faster through concurrent message passing.

All the results presented in this section were obtained from a simulator. In the following section, we return to the transmission time on any link as an exponential random variable with mean $1/\mu$ and we confirm the validity of the corresponding simulation results.

2.10 Analytic Results

The total delay (D) of a particular message is a random variable. The random variable for the time spent in the transmission queue, waiting for an available link, is called the waiting (W) delay. The transmission (T) delay is a random variable for the time a message spends en route from the sender to the receiver. The third random variable in the total delay, as discussed in Section 2.8, is the resequencing (R) delay. The total delay is then

$$D = W + T + R.$$

Once again, consider the system shown in Figure 2.9. The message arrival process is Poisson with rate λ. The service time, or transmission delay, is exponentially distributed with mean $1/\mu$:

$$T(x) = 1 - e^{-\mu x} \qquad x \geq 0.$$

The system has N identical links connecting two communicating processes. We assume there is an infinite number of messages to be transmitted. The system up to, but not including, the resequencing buffer is an $M/M/N$ queue. This subsystem will achieve steady state for $\rho = \frac{\lambda}{N\mu} < 1$.

Let p_n be the steady state probability of finding n messages in the subsystem. The state occupancy probabilities for the *M/M/N* queue at steady state are given by [Kle75]:

$$p_n = \begin{cases} p_0 \dfrac{(N\rho)^n}{n!} & n \leq N \\ \\ p_0 \dfrac{(N\rho)^n}{N!N^{n-N}} & n \geq N \end{cases} \tag{2.1}$$

where

$$p_0 \;=\; \left[\sum_{n=0}^{N-1} \frac{(N\rho)^n}{n!} + \left(\frac{(N\rho)^N}{N!}\right)\left(\frac{1}{1-\rho}\right)\right]^{-1}.$$

The expected total delay is given by:

$$\mathrm{E}[D] = \mathrm{E}[W] + \mathrm{E}[T] + \mathrm{E}[R]. \tag{2.2}$$

Independent of the communication paradigm, E[W] is given from *M/M/N* analysis [AR87],

$$\mathrm{E}[W] \;=\; \frac{p_0}{\lambda N!}\,\frac{\rho\,(N\rho)^N}{(1-\rho)^2},$$

and

$$\mathrm{E}[T] \;=\; \frac{1}{\mu}.$$

The resequencing delay, on the other hand, does depend on the communication paradigm. In the next four sections, we discuss the expected resequencing delay for different communication scenarios. The first section lists established results for the resequencing delay for a virtual circuit (VC). The next three sections derive the resequencing delay for the different batching scenarios discussed in Section 2.8.2.

2.10.1 Virtual Circuit Resequencing Delay

Analysis of the resequencing problem for virtual circuit communication has been considered in the literature by several authors. Yum and Ngai [YN86] studied the resequencing of messages in an $M/M/N$ queuing system with links of different speeds. Messages in this system were transmitted down the fastest available link. They found that resequencing delays increase as the variation in the speed of the links increase. In [AR87], Agrawal and Ramaswamy focused on the distributional aspects of the resequencing delay in an $M/M/N$ system. Chowdhury derived the distribution of the total delay [Cho89].

The expected resequencing delay of a message transmitted across a virtual circuit could be derived from the resequencing delay distribution. It is far simpler, however, to derive the expected delay by conditioning the derivation on the number of messages in transmission and then, subsequently, removing the condition [AR87]. We outline the approach of Agrawal and Ramaswamy since we use a similar argument. Suppose $n < N$ links are busy. When a message arrives, it is tagged and transmitted immediately. Let T_{n+1} represent the transmission time of this tagged message. Due to the memoryless property of the exponential transmission time distribution, the remaining transmission time for each message en route, at the instant of transmission of the tagged message, is exponentially distributed with mean $1/\mu$. Let t_k be the remaining transmission time of the message on the kth busy link ($k \leq n$). Thus, $\max(\mathrm{t}_1, \mathrm{t}_2, \ldots, \mathrm{t}_n, \mathrm{T}_{n+1})$ is the time until all $(n+1)$ messages en route have completed transmission.

Let $\mathrm{VC}R_n$ be the resequencing delay of the tagged message transmitted across a virtual circuit when n links are busy. Then

$$\mathrm{E}[\mathrm{VC}R_n] = \mathrm{E}[\max(\mathrm{t}_1, \mathrm{t}_2, \ldots, \mathrm{t}_n, \mathrm{T}_{n+1}) - \mathrm{T}_{n+1}]. \tag{2.3}$$

Equation (2.3) is the expected resequencing delay conditioned on the number of busy links.

If VCR is the unconditioned resequencing delay, then

$$\mathrm{E[VC}R] = \sum_{n=0}^{N-1} pb(n)\ \mathrm{E[VC}R_n], \tag{2.4}$$

where $pb(n)$ is the probability of finding n busy links upon the transmission of the tagged message. These probabilities can be obtained directly from the steady-state probability of state occupancy in an $M/M/N$ queue [AR87]:

$$\begin{aligned} pb(n) &= p_n, \\ &= p_0 \frac{(N\rho)^n}{n!}, \qquad\qquad 0 \le n \le N-2. \\ pb(N\text{-}1) &= \sum_{j=N-1}^{\infty} p_n, \\ &= p_0 \frac{(N\rho)^{N-1}}{(N-1)!(1-\rho)}. \end{aligned} \tag{2.5}$$

Using equations (2.3)—(2.5), Agrawal and Ramaswamy derive the expected resequencing delay in [AR87]:

$$\begin{aligned} \mathrm{E[VC}R] &= \sum_{n=0}^{N-1} pb(n)\ \mathrm{E}[\max(\mathrm{t}_1, \mathrm{t}_2, \ldots, \mathrm{t}_n, \mathrm{T}_{n+1}) - \mathrm{T}_{n+1}], \\ &= \frac{p_0}{\mu}\left[\sum_{n=0}^{N-2} \frac{(N\rho)^n}{n!} \sum_{k=2}^{n+1} \frac{1}{k} + \frac{(N\rho)^{N-1}}{(N-1)!(1-\rho)} \sum_{k=2}^{N} \frac{1}{k}\right]. \end{aligned} \tag{2.6}$$

Combining the expected waiting, transmission, and resequencing delays produces an equation for the expected total delay a message experiences across a virtual circuit. If VCD represents the total delay of a particular message transmitted across a virtual circuit, then

$$\mathrm{E[VC}D] = \frac{p_0}{\lambda N!} \frac{\rho\,(N\rho)^N}{(1-\rho)^2} + \frac{1}{\mu} + \frac{p_0}{\mu}\left[\sum_{n=0}^{N-2} \frac{(N\rho)^n}{n!} \sum_{k=2}^{n+1} \frac{1}{k} + \frac{(N\rho)^{N-1}}{(N-1)!(1-\rho)} \sum_{k=2}^{N} \frac{1}{k}\right]. \tag{2.7}$$

Results from this equation are within the confidence interval of the simulation results shown in Figures 2.13 and 2.14.

In [AR87], aspects of the resequencing distribution are discussed. They analytically show that E[VCR] always increases as

(1) N increases,

(2) λ increases,

(3) μ decreases.

Let us assume that ρ increases by either keeping λ fixed and decreasing μ or by keeping μ fixed and increasing λ. Then, E[VCR] monotonically increases as N or ρ increases. Since utilization cannot increase past 1.0, equation (2.6) reaches its limit when $\rho = 1.0$ and all other variables are fixed. When all variables are fixed and N is increased, E[VCR] converges. To calculate the asymptote of the convergence, we consider the closed-form solution for a system with an infinite number of links. In [Kle75], we obtain the probability that n links are busy in an $M/M/\infty$ system as

$$p_n = \left(\frac{\lambda}{\mu}\right)^n \frac{1}{n!} e^{-\lambda/\mu}. \tag{2.8}$$

Using p_n, Agrawal and Ramaswamy derive E[VCR_∞], the expected resequencing delay of a particular message transmitted across a virtual circuit in a system with an infinite number of links [AR87]:

$$\begin{aligned} \mathrm{E}[\mathrm{VC}R_\infty] &= \sum_{n=0}^{\infty} p_n \, \mathrm{E}[\mathrm{VC}R_n], \\ &= \frac{1}{\mu}\left[\sum_{n=1}^{\infty}(-1)^{n-1}\left(\frac{\lambda}{\mu}\right)^n \frac{1}{n(n+1)!}\right]. \end{aligned} \tag{2.9}$$

2.10.2 Forward Flush Batching Scenario

In this section, we consider the resequencing delay for the transmission of the forward flush batching scenario. Suppose $\mathrm{FC}R_n$ is the resequencing delay of a message just arrived to find n busy links on an F-channel. Recall Figure 2.10; this graph illustrates that the predecessor set of an ordinary message is empty. The resequencing delay of an ordinary message is thus zero,

$$\mathrm{FC}R_n^{\mathrm{Ord}} = 0.$$

The resequencing delay of a forward flush message is more interesting. When a message of this type is tagged for transmission, every message previously transmitted will be in the predecessor set of this tagged forward flush message. Hence, the delivery of a forward flush message must wait for the delivery of every message currently in transmission, i.e., all n messages. Therefore,

$$\mathrm{FC}R_n^{\mathrm{FF}} = \max(\mathrm{t_1}, \mathrm{t_2}, \ldots, \mathrm{t_n}, \mathrm{T_{n+1}}) - \mathrm{T_{n+1}}.$$

If $\mathrm{FF}R_n$ is the resequencing delay of a tagged message in the forward flush batching scenario conditioned on n busy links, then

$$\begin{aligned}\mathrm{FF}R_n &= \left(\frac{B}{B+1}\right)\mathrm{FC}R_n^{\mathrm{Ord}} + \left(\frac{1}{B+1}\right)\mathrm{FC}R_n^{\mathrm{FF}}, \\ &= \left(\frac{1}{B+1}\right)\mathrm{FC}R_n^{\mathrm{FF}},\end{aligned}$$

where B is the batch size.

Following the derivation of the expected resequencing delay for virtual circuit communication, we derive the expected resequencing delay of the forward flush batching scenario conditioned on the number of busy links in the system and then remove the conditioning.

Equation 2.3 is the expected resequencing delay of a message that must wait for the delivery of every message currently in transmission. Therefore,

$$\mathrm{E}[\mathrm{FF}R_n] = \left(\frac{1}{B+1}\right)(\mathrm{E}[\max(t_1, t_2, \ldots t_n, T_{n+1})] - \mathrm{E}[T_{n+1}]).$$

Since $T(x)$ is the c.d.f. of message transmission time, then

$$\mathrm{E}[\mathrm{FF}R_n] = \left(\frac{1}{B+1}\right)\left(\int_0^\infty [1-(T(x))^{n+1}]dx - \frac{1}{\mu}\right).$$

Substituting z for $T(x)$, we obtain

$$\mathrm{E}[\mathrm{FF}R_n] = \left(\frac{1}{B+1}\right)\left(\frac{1}{\mu}\int_0^1 \frac{(1-z^{n+1})}{1-z}dz - \frac{1}{\mu}\right),$$

which, upon integration, is equivalent to

$$\mathrm{E}[\mathrm{FF}R_n] = \left(\frac{1}{B+1}\right)\left(\frac{1}{\mu}\sum_{k=1}^{n+1}\frac{1}{k} - \frac{1}{\mu}\right).$$

Using $pb(n)$, from (2.5), we remove the conditioning to obtain the expected resequencing delay of a particular message in the forward flush batching scenario.

$$\begin{aligned}
\mathrm{E}[\mathrm{FF}R] &= \sum_{n=0}^{N-1} pb(n)\ \mathrm{E}[\mathrm{FF}R_n], \\
&= \left(\frac{1}{B+1}\right)\left[\sum_{n=0}^{N-2} pb(n)\left(\frac{1}{\mu}\sum_{k=2}^{n+1}\frac{1}{k}\right) + pb(N\text{-}1)\left(\frac{1}{\mu}\sum_{k=2}^{N}\frac{1}{k}\right)\right], \\
&= \left(\frac{1}{B+1}\right)\frac{p_0}{\mu}\left[\sum_{n=0}^{N-2}\frac{(N\rho)^n}{n!}\sum_{k=2}^{n+1}\frac{1}{k} + \frac{(N\rho)^{N-1}}{(N-1)!(1-\rho)}\sum_{k=2}^{N}\frac{1}{k}\right]. \qquad (2.10)
\end{aligned}$$

Therefore,

$$\mathrm{E}[\mathrm{FF}R] = \left(\frac{1}{B+1}\right)\mathrm{E}[\mathrm{VC}R]. \qquad (2.11)$$

Equation 2.11 is intuitive; B out of $(B+1)$ messages in the system do not have any resequencing delay and 1 out of $(B+1)$ messages in the system have a resequencing delay as if the message were transmitted across a virtual circuit.

Combining the expected waiting, transmission, and resequencing delays yields the total delay a message in the forward flush batching scenario may expect:

$$\mathrm{E}[\mathrm{FF}D] = \frac{p_0}{\lambda N!}\frac{\rho\,(N\rho)^N}{(1-\rho)^2} + \frac{1}{\mu} + \left(\frac{1}{B+1}\right)\sum_{n=0}^{N-1} pb(n)\left(\frac{1}{\mu}\sum_{k=2}^{n+1}\frac{1}{k}\right). \qquad (2.12)$$

The result of plotting this equation is within the confidence interval of the corresponding simulation result and illustrated in Figure 2.16.

We consider the expected resequencing delay of a message in the forward flush batching scenario as the three parameters (B, ρ, and N) individually increase. Equation (2.10) shows that E[FFR] monotonically decreases when B increases and all other variables are fixed. In [AR87], Agrawal and Ramaswamy proved that $pb_1(n)$ is *stochastically* larger than $pb_2(n)$ if ρ in $pb_1(n)$ is smaller than ρ in $pb_2(n)$. (Recall that we assume ρ increases by keeping either λ or μ fixed.) That is,

$$\sum_{n=k}^{N-1} pb_1(n) \leq \sum_{n=k}^{N-1} pb_2(n) \quad \text{for } 0 \leq k \leq N-1.$$

Since E[FFR_n] increases when μ is decreased or n is increased, we conclude that E[FFR] monotonically increases as ρ increases and all other variables are fixed. Furthermore, E[FFR] reaches a bound when $\rho = 1.0$. Lastly, we consider E[FFR] when N is increased. For the virtual circuit communication paradigm, Agrawal and Ramaswamy proved that the expected resequencing delay monotonically increases as N increases. Consider this fact intuitively; if N increases and all other variables are fixed, more messages are on the links at any given time and, hence, there is more opportunity for the messages to arrive out of order. From equation (2.11), we conclude the mean message delay in the forward flush batching scenario monotonically increases when N increases as well. E[FFR] converges to the expected resequencing delay for a message in an $M/M/\infty$ system. Using (2.8), the probability that n links are busy in an $M/M/\infty$ system, we derive the expected resequencing as

$$\begin{aligned} \mathrm{E}[\mathrm{FF}R_\infty] &= \sum_{n=0}^{\infty} p_n \ \mathrm{E}[\mathrm{FF}R_n], \\ &= \left(\frac{1}{B+1}\right) \sum_{n=0}^{\infty} p_n \ \frac{1}{\mu} \sum_{k=2}^{n+1} \frac{1}{k}. \end{aligned}$$

From the derivation of E[FFR_n], we know that $\sum_{k=2}^{n+1} \frac{1}{k}$ is equivalent to $\int_0^1 \frac{x}{1-x}(1-x^n)dx$. Therefore,

$$\mathrm{E}[\mathrm{FF}R_\infty] = \left(\frac{1}{B+1}\right) \sum_{n=1}^{\infty} p_n \ \frac{1}{\mu} \int_0^1 \frac{x}{1-x}(1-x^n)dx.$$

The expected value then becomes

$$\mathrm{E}[\mathrm{FF}R_\infty] = \left(\frac{1}{B+1}\right) \frac{1}{\mu} \int_0^1 \frac{x}{1-x}(1-e^{-(1-x)\lambda/\mu})dx$$

when the infinite series is expanded. Substituting $y = (1-x)$ and the power series for e^x, we obtain

$$\begin{aligned} \mathrm{E}[\mathrm{FF}R_\infty] &= \left(\frac{1}{B+1}\right) \frac{1}{\mu} \int_0^1 \left[\sum_{n=1}^{\infty} (-1)^{n-1} \left(\frac{\lambda}{\mu}\right)^n \frac{y^{n-1}}{n!} - \sum_{n=1}^{\infty} (-1)^{n-1} \left(\frac{\lambda}{\mu}\right)^n \frac{y^n}{n!} \right] dy, \\ &= \left(\frac{1}{B+1}\right) \frac{1}{\mu} \sum_{n=1}^{\infty} (-1)^{n-1} \left(\frac{\lambda}{\mu}\right)^n \frac{1}{n(n+1)!} \qquad (2.13) \end{aligned}$$

upon integration and simplification. Again,

$$\mathrm{E}[\mathrm{FF}R_\infty] = \left(\frac{1}{B+1}\right) \mathrm{E}[\mathrm{VC}R_\infty]. \qquad (2.14)$$

2.10.3 Two-way Flush Batching Scenario

Figure 2.11 shows the two-way flush batching scenario. Similar to a forward flush message, the delivery of a two-way flush message must wait for the delivery of every message currently in transmission. If a tagged two-way flush message is ready for transmission and n links are busy, then

$$\mathrm{FC}R_n^{\mathrm{2F}} = \max(\mathrm{t}_1, \mathrm{t}_2, \ldots, \mathrm{t}_\mathrm{n}, \mathrm{T}_{\mathrm{n}+1}) - \mathrm{T}_{\mathrm{n}+1}.$$

Consider the transmission of the jth ordinary message. If x messages from the predecessor set of this tagged ordinary message are in transit, then

$$\mathrm{FC}R_n^{\mathrm{Ord},\,j} = \begin{cases} 0 & \text{if } x = 0 \\ \max(\mathrm{t}_1, \mathrm{T}_{\mathrm{n}+1}) - \mathrm{T}_{\mathrm{n}+1}, & \text{if } x = 1 \\ \max(\mathrm{t}_1, \mathrm{t}_2, \mathrm{T}_{\mathrm{n}+1}) - \mathrm{T}_{\mathrm{n}+1}, & \text{if } x = 2 \\ \vdots & \vdots \\ \max(\mathrm{t}_1, \mathrm{t}_2, \ldots, \mathrm{t}_\mathrm{n}, \mathrm{T}_{\mathrm{n}+1}) - \mathrm{T}_{\mathrm{n}+1}, & \text{if } x = \mathrm{n} \end{cases}$$

To calculate $\mathrm{FC}R_n^{\mathrm{Ord},\,j}$, we require the probability that x messages are in transit from the predecessor set of the jth ordinary message. If i ordinary messages from the batch containing j are in transit, then $i = n - x$. Instead of calculating the probability that x messages are in transit from the predecessor set given n links are busy, we calculate, upon the transmission of the jth ordinary message,

$$\mathrm{pords}_j(i \mid n) = P\{i \text{ ordinary messages in transit from the current batch} \mid n \text{ busy links}\}.$$

We re-write $\mathrm{FC}R_n^{\mathrm{Ord},\,j}$ in terms of the number of ordinary messages in transit from the current batch given n links are busy:

$$\begin{aligned}
\mathrm{FC}R_n^{\mathrm{Ord},\,j} &= \begin{cases} \mathrm{pords}_j(\min(j-1,n) \mid n)\ (\max(\mathrm{t}_1,\mathrm{t}_2,\ldots,\mathrm{t}_{\mathrm{n}-\min(\mathrm{j}-1,\mathrm{n})},\mathrm{T}_{\mathrm{n}+1})-\mathrm{T}_{\mathrm{n}+1}) \\ \vdots \\ +\mathrm{pords}_j(\mathit{2} \mid n)\ (\max(\mathrm{t}_1,\mathrm{t}_2,\ldots,\mathrm{t}_{\mathrm{n}-2},\mathrm{T}_{\mathrm{n}+1})-\mathrm{T}_{\mathrm{n}+1}) \\ +\mathrm{pords}_j(\mathit{1} \mid n)\ (\max(\mathrm{t}_1,\mathrm{t}_2,\ldots,\mathrm{t}_{\mathrm{n}-1},\mathrm{T}_{\mathrm{n}+1})-\mathrm{T}_{\mathrm{n}+1}) \\ +\mathrm{pords}_j(\mathit{0} \mid n)\ (\max(\mathrm{t}_1,\mathrm{t}_2,\ldots,\mathrm{t}_{\mathrm{n}},\mathrm{T}_{\mathrm{n}+1})-\mathrm{T}_{\mathrm{n}+1}) \end{cases} \\
&= \sum_{i=0}^{\min(j-1,n)} \mathrm{pords}_j(i \mid n)\ (\max(\mathrm{t}_1,\mathrm{t}_2,\ldots,\mathrm{t}_{\mathrm{n}-\mathrm{i}},\mathrm{T}_{\mathrm{n}+1})-\mathrm{T}_{\mathrm{n}+1}).
\end{aligned}$$

Following the derivation of the expected resequencing delay in the two previous sections, we first consider the expected resequencing delay of the two-way flush batching scenario ($\mathrm{E}[2\mathrm{F}R]$) conditioned on the number of busy links in the $M/M/N$ system.

$$\begin{aligned}
\mathrm{E}[2\mathrm{F}R_n] &= \left(\frac{1}{B+1}\right)\sum_{j=1}^{B}\mathrm{E}[\mathrm{FC}R_n^{\mathrm{Ord},\,j}] + \left(\frac{1}{B+1}\right)\mathrm{E}[\mathrm{FC}R_n^{\mathrm{2F}}], \\
&= \left(\frac{1}{B+1}\right)\sum_{j=1}^{B}\sum_{i=0}^{\min(j-1,n)} \mathrm{pords}_j(i \mid n)\ \mathrm{E}[\max(\mathrm{t}_1,\mathrm{t}_2,\ldots,\mathrm{t}_{\mathrm{n}-\mathrm{i}},\mathrm{T}_{\mathrm{n}+1})-\mathrm{T}_{\mathrm{n}+1}] \\
&\qquad + \left(\frac{1}{B+1}\right)\mathrm{E}[\ \max(\mathrm{t}_1,\mathrm{t}_2,\ldots,\mathrm{t}_{\mathrm{n}},\mathrm{T}_{\mathrm{n}+1})-\mathrm{T}_{\mathrm{n}+1}], \\
&= \left(\frac{1}{B+1}\right)\left[\sum_{j=1}^{B}\sum_{i=0}^{\min(j-1,n)} \mathrm{pords}_j(i \mid n)\left(\frac{1}{\mu}\sum_{k=2}^{n-i+1}\frac{1}{k}\right) + \left(\frac{1}{\mu}\sum_{k=2}^{n+1}\frac{1}{k}\right)\right].
\end{aligned}$$

The expected resequencing delay in the two-way flush batching scenario is then

$$\begin{aligned}
\mathrm{E}[2\mathrm{F}R] &= \sum_{n=0}^{N-1} pb(n)\ \mathrm{E}[2\mathrm{F}R_n], \\
&= \left(\frac{1}{B+1}\right)\left[\sum_{j=1}^{B}\sum_{n=0}^{N-1} pb(n) \sum_{i=0}^{\min(j-1,n)} \mathrm{pords}_j(i \mid n)\left(\frac{1}{\mu}\sum_{k=2}^{n-i+1}\frac{1}{k}\right)\right. \\
&\qquad \left. + \sum_{n=0}^{N-1} pb(n)\left(\frac{1}{\mu}\sum_{k=2}^{n+1}\frac{1}{k}\right)\right]. \qquad (2.15)
\end{aligned}$$

The expected total delay a message from the two-way flush batching scenario experiences is, as before, the combination of the three expected delays:

$$\begin{aligned}\mathrm{E}[2\mathrm{F}D] &= \frac{p_0}{\lambda N!}\frac{\rho\,(N\rho)^N}{(1-\rho)^2}+\frac{1}{\mu}+\left(\frac{1}{B+1}\right)\left[\sum_{j=1}^{B}\sum_{n=0}^{N-1} pb(n)\sum_{i=0}^{\min(j-1,n)}\mathrm{pords}_j(i\mid n)\left(\frac{1}{\mu}\sum_{k=2}^{n-i+1}\frac{1}{k}\right)\right.\\ &\qquad\left.+\sum_{n=0}^{N-1} pb(n)\left(\frac{1}{\mu}\sum_{k=2}^{n+1}\frac{1}{k}\right)\right]. \qquad (2.16)\end{aligned}$$

We consider E[2FR] as B, ρ, and N individually increase. Equation (2.15) shows that E[2FR] monotonically decreases when B increases and all other variables are fixed. Now consider E[2FR] when ρ increases and all other variables are fixed. We know, from the previous section, that $pb(n)$ stochastically increases as ρ increases. Furthermore, $(\frac{1}{\mu}\sum_{k=2}^{n-i+1}\frac{1}{k})$ increases as μ decreases or n increases. Therefore, to verify E[2FR] monotonically increases as ρ increases, we need to show that $\mathrm{pords}_j(i\mid n)$ stochastically increases as ρ increases. That is, if ρ is higher in $\mathrm{pords}_j(i\mid n)_2$ than ρ in $\mathrm{pords}_j(i\mid n)_1$, we need to show that

$$\sum_{n=k}^{N-1}\sum_{i=0}^{\min(j-1,n)}\mathrm{pords}_j(i\mid n)_1 \le \sum_{n=k}^{N-1}\sum_{i=0}^{\min(j-1,n)}\mathrm{pords}_j(i\mid n)_2 \quad \text{for } 0\le k\le N-1.$$

This result is clear when we consider that $\sum_{i=0}^{\min(j-1,n)}\mathrm{pords}_j(i\mid n) = p_n$. Hence, as ρ increases, the expected resequencing delay in the two-way flush batching scenario monotonically increases. Furthermore, E[2FR] has a bound when $\rho = 1.0$. Lastly, we consider E[2FR] when N is increased. Intuitively, if N increases and all other variables are fixed, there is more opportunity for the messages to arrive out of order. Hence, the expected value monotonically increases as N increases. For validation, consider equation (2.15). If N is smaller in $pb_1(n)$ than N in $pb_2(n)$, then

$$\sum_{n=k}^{N-1} pb_1(n) \le \sum_{n=k}^{N-1} pb_2(n) \quad \text{for } 0\le k\le N-1.$$

Now consider $\mathrm{pords}_j(i\mid n)$. Since ρ is fixed, $\mathrm{pords}_j(i\mid n)$ for a given N is identical to

$\text{pords}_j(i \mid n)$ for a higher N. In addition, $(\frac{1}{\mu}\sum_{k=2}^{n-i+1}\frac{1}{k})$ increases as N increases. Therefore, equation (2.15) monotonically increases as N increases. $\mathrm{E}[2\mathrm{F}R]$ converges to the expected resequencing delay for a message in a system with an infinite number of links. The probability that n links are busy in an $M/M/\infty$ system is from (2.8).

$$\begin{aligned} \mathrm{E}[2\mathrm{F}R_\infty] &= \sum_{n=0}^{\infty} p_n \, \mathrm{E}[2\mathrm{F}R_n], \\ &= \left(\frac{1}{B+1}\right)\left[\sum_{j=1}^{B}\sum_{n=1}^{\infty} p_n \sum_{i=0}^{\min(j-1,n)} \text{pords}_j(i \mid n)\left(\frac{1}{\mu}\sum_{k=2}^{n-i+1}\frac{1}{k}\right)\right. \\ &\qquad \left. + \frac{1}{\mu}\sum_{n=1}^{\infty}(-1)^{n-1}\left(\frac{\lambda}{\mu}\right)^n \frac{1}{n(n+1)!}\right]. \end{aligned} \tag{2.17}$$

All of the derivations in this section require the probability that, upon the transmission of the jth ordinary message, i ordinary messages are in transit from the current batch given n links are busy: $\text{pords}_j(i \mid n)$. We know that

$$\sum_{i=0}^{\min(j-1,n)} \text{pords}_j(i \mid n) = 1$$

and

$$\text{pords}_j(i \mid n) = \frac{\text{pords}_j(i \wedge n)}{\sum_{i=0}^{j-1}\text{pords}_j(i \wedge n)}$$

where

$$\text{pords}_j(i \wedge n) = P\{i \text{ ordinary messages in transit from current batch } \wedge n \text{ busy links}\}.$$

We first consider $\text{pords}_j(i \wedge n)$ in the $M/M/\infty$ system, and then restrict the system to a finite number of links. Given a system with an infinite number of links, we consider the probability that a message previously transmitted is in transit when another message begins transmission. Suppose the inter-arrival interval preceding the transmission of m_i is A_i and the transmission time of m_i is T_i. Then, as illustrated in Figure 2.27, $P\{A_2 < T_1\}$ is the probability that m_1 is in transit when m_2 begins transmission. Furthermore, $P\{A_2 + A_3 < T_1\}$ is the probability that m_1 is in transit when m_3 begins transmission.

Since the inter-arrival and transmission times are independent exponential random variables, with means $1/\lambda$ and $1/\mu$ respectively, we compute the above two probabilities as

$$\begin{aligned} P\{A_2 < T_1\} &= \int_0^\infty P\{A_2 < T_1 \mid T_1 = x\}\mu e^{-\mu x}dx, \\ &= \frac{\lambda}{\lambda + \mu} \end{aligned}$$

and

$$\begin{aligned} P\{A_2 + A_3 < T_1\} &= \int_0^\infty P\{A_2 + A_3 < T_1 \mid T_1 = x\}\mu e^{-\mu x}dx, \\ &= \left(\frac{\lambda}{\lambda + \mu}\right)^2. \end{aligned}$$

Generalizing, we obtain the probability that the nth message previously transmitted is in transit when a new message is transmitted [Ros73]:

$$\begin{aligned} P\{A_2 + A_3 + \ldots + A_n < T_1\} &= \left(\frac{\lambda}{\lambda + \mu}\right)^n, \\ P\{nA < T_1\} &= \alpha^n \end{aligned} \tag{2.18}$$

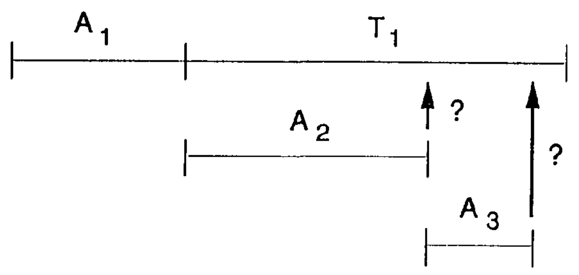

Figure 2.27: Inter-arrival and Transmission Times: $M/M/\infty$

where $\alpha = \frac{\lambda}{\lambda+\mu}$.

Returning to $\text{pords}_j(i \wedge n)$, let us consider two specific examples. Suppose the jth ordinary message from a batch is tagged and ready for transmission. Then

$$\begin{aligned} p_0 &= \text{pords}_j(0 \wedge 0), \\ &= P\{T < A \wedge T < 2A \wedge T < 3A \wedge \ldots\}; \end{aligned}$$

i.e., no message previously transmitted is in transit when the jth ordinary message is transmitted. Due to equation 2.18, the value of this probability would be trivial to obtain if the individual components within the probability were independent. We test for dependence by whether

$$P\{T_1 < A_2 + A_3 \mid T_2 < A_3\} = P\{T_1 < A_2 + A_3\}.$$

Consider S, the set of all possible values in the $M/M/\infty$ system for (T_1, A_2, A_3, T_2). We define the events E_1, E_2, and E_3 as the following subsets of the sample space S.

$$\begin{aligned} E_1 &= \{x = (T_1, A_2, A_3, T_2) \mid T_1 < A_2 \ \wedge \ T_1 < A_3 \ \wedge \ T_2 < A_3\}. \\ E_2 &= \{x = (T_1, A_2, A_3, T_2) \mid T_1 < A_2 \ \wedge \ T_1 > A_3 \ \wedge \ T_1 < A_2 + A_3 \ \wedge \ T_2 < A_3\}. \\ E_3 &= \{x = (T_1, A_2, A_3, T_2) \mid T_1 > A_2 \ \wedge \ T_1 < A_3 \ \wedge \ T_1 < A_2 + A_3 \ \wedge \ T_2 < A_3\}. \end{aligned}$$

The union of E_1, E_2, and E_3, defined as E, is all possible outcomes of $T_1 < A_2 + A_3$ given $T_2 < A_3$. We now define the events F_1, F_2, and F_3 as the following subsets in S.

$$\begin{aligned} F_1 &= \{x = (T_1, A_2, A_3, T_2) \mid T_1 < A_2 \ \wedge \ T_1 < A_3\}. \\ F_2 &= \{x = (T_1, A_2, A_3, T_2) \mid T_1 < A_2 \ \wedge \ T_1 > A_3 \ \wedge \ T_1 < A_2 + A_3\}. \\ F_3 &= \{x = (T_1, A_2, A_3, T_2) \mid T_1 > A_2 \ \wedge \ T_1 < A_3 \ \wedge \ T_1 < A_2 + A_3\}. \end{aligned}$$

The union of F_1, F_2, and F_3, defined as F, is all possible outcomes of $T_1 < A_2 + A_3$ without the constraint $T_2 < A_3$. When we consider the probabilities of E and F, we conclude that the dependence test fails. For instance, when $T_1 < A_2$ and $T_1 > A_3$, E implies that $T_2 < T_1$. F_2 makes no such implication. That is, F represents possible outcomes that E cannot represent; hence, the probability of F will be greater than the probability of E. Therefore,

$$P\{T_1 < A_2 + A_3 \mid T_2 < A_3\} \quad \neq \quad P\{T_1 < A_2 + A_3\}$$

and the individual components within $\text{pords}_j(i \wedge n)$ are dependent.

We can, however, obtain an approximation for $\text{pords}_j(i \wedge n)$ denoted $\text{pords}^*_j(i \wedge n)$, by assuming they are independent. Thus,

$$\begin{aligned} \text{pords}^*_j(0 \wedge 0) &= P\{T < A\}P\{T < 2A\}P\{T < 3A\}\ldots, \\ &= (1-\alpha)(1-\alpha^2)(1-\alpha^3)\ldots \end{aligned}$$

For another example, consider the transmission of the third ordinary message in a batch. Then

$$\begin{aligned} \text{pords}^*_3(0 \wedge 1) = {} & (1-\alpha)(1-\alpha_2)\,\alpha^3\,(1-\alpha^4)(1-\alpha^5)(1-\alpha^6)\ldots \\ & +(1-\alpha)(1-\alpha_2)(1-\alpha^3)\,\alpha^4\,(1-\alpha^5)(1-\alpha^6)\ldots \\ & +(1-\alpha)(1-\alpha_2)(1-\alpha^3)(1-\alpha^4)\,\alpha^5\,(1-\alpha^6)\ldots \\ & \vdots \end{aligned}$$

and

$$\begin{aligned} \text{pords}^*_3(1 \wedge 1) = {} & \alpha\,(1-\alpha_2)(1-\alpha^3)(1-\alpha^4)(1-\alpha^5)(1-\alpha^6)\ldots \\ & +(1-\alpha)\,\alpha_2\,(1-\alpha^3)(1-\alpha^4)(1-\alpha^5)(1-\alpha^6)\ldots \end{aligned}$$

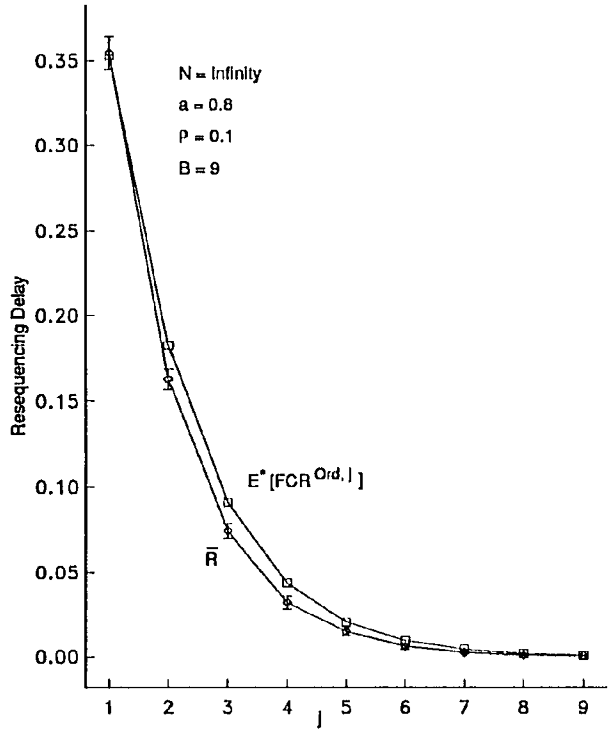

Figure 2.28: Analysis vs. Simulation Results: $M/M/\infty$

The above two examples indicate the method to obtain an approximation of $\text{pords}_j(i \wedge n)$. $\text{pords}^*_j(i \wedge n)$ is a summation of probabilities. Each probability is a product of two products. The first product is the probability that i ordinary messages from the jth batch are in transit; the second product is the probability that $n - i$ messages not from the jth batch are in transit. The two examples of $\text{pords}^*_j(i \wedge n)$ above were chosen due to their simplicity. That is, the number of possible combinations for the first product is $\binom{j-1}{i}$; the number of possible combinations for the second product is either one or ∞. For simplification in the examples above, only one combination is possible for the first or second products or both.

In Figure 2.28, we plot the approximate expected resequencing delay for nine ordinary messages in a batch, $\text{E}^*[\text{FC}^{\text{Ord},j}]$, when $a = \frac{\lambda}{\mu} = 0.8$; a represents the average number of busy links in an $M/M/\infty$ system. We compare this result to $\bar{R}$ for a simulated 8-link system with $\lambda = 0.8$ and $\mu = 1.0$. In the simulated system, the average number of busy links is 0.8 as well. The maximum number of busy links in the analysis is eight. We do not plot the associated resequencing delay for a two-way flush message; this value is equivalent to the value of the first ordinary message in a batch for both E[FC$^{\mathrm{Ord},1}$] and $\bar{R}$. As the figure illustrates (although we approximate $\mathrm{pords}_j(i \wedge n)$ with $\mathrm{pords}^*_j(i \wedge n)$), the results from the analytical equations match the simulation results. In the simulation, the mean resequencing delay is between 0.0968 and 0.1014 with 95% confidence. The result from the approximate analysis is 0.1059.

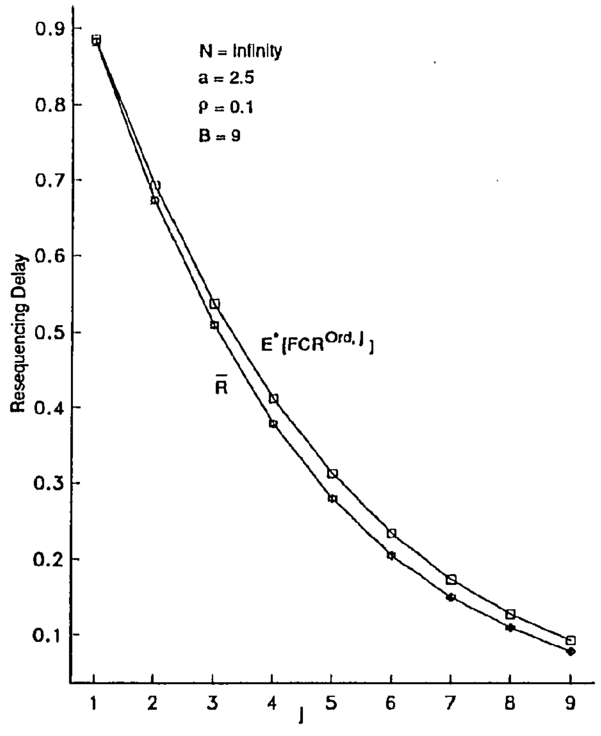

Figure 2.29: Analysis vs. Simulation Results: $M/M/\infty$

Figure 2.29 plots the approximate expected resequencing delay for nine ordinary messages when the average number of busy links is 2.5. We compare the result to $\bar{R}$ from a simulated 25-link system with $\lambda = 2.5$ and $\mu = 1.0$. The average number of busy links in the simulation is also 2.5; the maximum number of busy links from the analysis is 13. In this situation, the mean resequencing delay from the simulator is between 0.4114 and

0.4173. The analysis approximates the expected resequencing delay at 0.4357.

In a system with a finite number of links, we again consider the probability that a message previously transmitted is in transit when another message begins transmission. As before, A_i is the inter-arrival interval preceding the transmission of m_i; T_i is its transmission time. Q_i represents the queueing delay for message m_i. Figure 2.30 illustrates that the

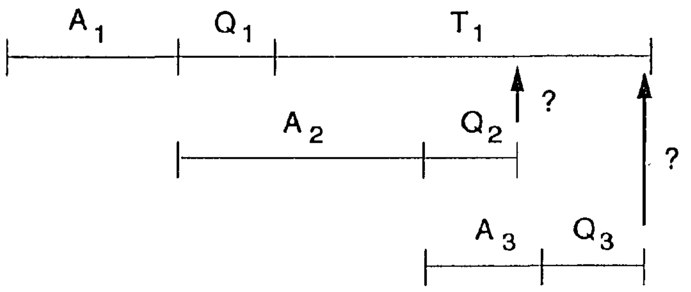

Figure 2.30: Inter-arrival and Transmission Times: $M/M/N$

probability that m_1 is in transit upon the transmission of m_2 is $P\{A_2+Q_2 < Q_1+T_1\}$. The probability that m_1 is in transit when m_3 begins transmission is $P\{A_2+A_3+Q_3 < Q_1+T_1\}$.

Unlike a system with an infinite number of links, these two probabilities are difficult to obtain We do not examine the derivations of these probabilities, however, for what we require is $P\{A_2 + Q_2 < Q_1 + T_1$ **and** $A_2 + A_3 + Q_3 < Q_1 + T_1\}$. Let us consider a specific example. Suppose the jth ordinary message from a batch is tagged and ready for transmission. Then

$$\begin{aligned} pb(0) &= \text{pords}_j(0 \wedge 0), \\ &= P\{Q+T < A+Q \wedge Q+T < 2A+Q \wedge Q+T < 3A+Q \wedge \ldots\}; \end{aligned}$$

i.e., no message previously transmitted is in transit when the jth ordinary message is transmitted.

As in the case of a system with an infinite number of links, the conditions within the probability are not independent. We, therefore, approximate the probability by assuming independence; e.g.,

$$\text{pords}_j^*(0 \wedge 0) = P\{Q+T < A+Q\}P\{Q+T < 2A+Q\}P\{Q+T < 3A+Q\}\ldots.$$

Because there is no simple derivation for $P\{Q+T < nA+Q\}$, we approximate $\text{pords}_j^*(i \wedge n)$ one step further. For a given utilization, the queueing delays for any two messages will, most likely, be close in value. Therefore, we further approximate $\text{pords}_j^*(i \wedge n)$ as we did in the case of a system with an infinite number of links by subtracting out the queueing delays:

$$\begin{aligned}\text{pords}_j^*(0 \wedge 0) &= P\{T < A\}P\{T < 2A\}P\{T < 3A\}\ldots, \\ &= (1-\alpha)(1-\alpha^2)(1-\alpha^3)\ldots\end{aligned}$$

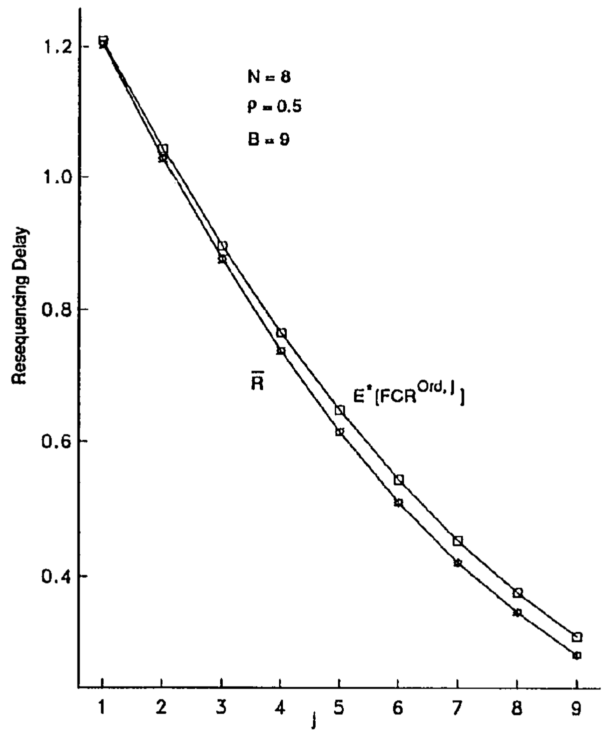

Figure 2.31: Analysis vs. Simulation Results: *M/M/8*

In Figure 2.31, we plot the approximate expected resequencing delay for nine ordinary messages, $E^*[FC^{Ord,j}]$, in an *M/M/8* system when $\lambda = 4.0$ and $\mu = 1.0$. We compare this result to the corresponding simulation result; the two results validate one another. The mean resequencing delay, $\bar{R}$, from the simulation is between 0.7177 and 0.7264; the result from the approximate expected value is 0.7450.

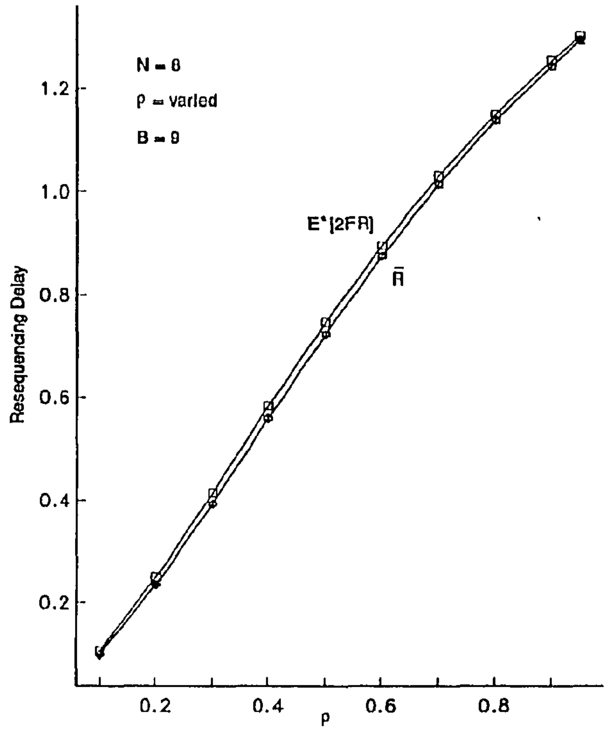

Figure 2.32: Analysis vs. Simulation Results: *M/M/8*

Figure 2.32 plots the two expected resequencing delays of the two-way flush batching scenario, simulation and approximate analysis, for an 8-link system. Although utilization varies from no queueing delay to system saturation, the simulation and approximate analysis results continue to validate one another.

One interesting phenomenon of the two-way flush batching scenario occurs when the batch size is increased. The *j*th ordinary message in an F-channel application with batch size *B* has the same expected resequencing delay as the *j*th ordinary message in another

F-channel application *regardless* of the batch size. If the batch size is increased by one, the total expected resequencing delay is the previous batch's total expected resequencing delay with the consideration of the additional ordinary message in each batch. That is,

$$\mathrm{E}[2\mathrm{F}R_{B+1}] = \left(\frac{B+1}{B+2}\right)\mathrm{E}[2\mathrm{F}R_B] + \left(\frac{1}{B+2}\right)\mathrm{E}[\mathrm{FC}^{\mathrm{Ord},\,B+1}]. \tag{2.19}$$

2.10.4 Backward Flush Batching Scenario

Recall Figure 2.12, the backward flush batching scenario. The delivery of a backward flush message is based upon the delivery of the previous backward flush point, which, in this situation, is the previous backward flush message transmitted. If a tagged backward flush message is ready for transmission, n links are busy, and the number of backward flush messages currently transmitting is i, then

$$\mathrm{FC}R_n^{\mathrm{BF}} = \begin{cases} 0 & \text{if } i = 0 \\ \max(\mathrm{t}_1, \mathrm{T}_{\mathrm{n}+1}) - \mathrm{T}_{\mathrm{n}+1}, & \text{if } i = 1 \\ \max(\mathrm{t}_1, \mathrm{t}_2, \mathrm{T}_{\mathrm{n}+1}) - \mathrm{T}_{\mathrm{n}+1}, & \text{if } i = 2 \\ \vdots & \vdots \\ \max(\mathrm{t}_1, \mathrm{t}_2, \ldots, \mathrm{t}_{\mathrm{n}}, \mathrm{T}_{\mathrm{n}+1}) - \mathrm{T}_{\mathrm{n}+1}, & \text{if } i = \mathrm{n} \end{cases}$$

The criteria for delivery of an ordinary message are identical to the criteria for delivery of a backward flush message. If i is the number of messages in transit from the predecessor set of the tagged ordinary message, i.e., the number of backward flush messages in transit, then

$$\mathrm{FC}R_n^{\mathrm{Ord}} = \mathrm{FC}R_n^{\mathrm{BF}}.$$

In order to calculate the expected resequencing delay of this batching scenario, we require the probability that i backward flush messages are in transit when a tagged message is transmitted. This probability depends upon the number of message transmissions since the transmission of the previous backward flush message. For a backward flush message, $(B + 1)$ messages have been transmitted since the transmission of the previous backward flush message; for the jth ordinary message in a batch, j messages have been transmitted since the transmission of the previous backward flush. The probability that x backward flush messages are in transit at the transmission of the tagged message is based upon c, the number of message transmissions since the transmission of the previous backward flush message, and n, the number of busy links. If

$$\mathrm{pbf}_c(i \mid n) = P\{i \text{ backward flush messages in transit} \mid n \text{ busy links}\}$$

then

$$\begin{aligned}
\mathrm{FC}R_n^{\mathrm{BF},\,B+1} &= \left\{\begin{array}{l}
\mathrm{pbf}_{B+1}(0 \mid n)\; 0 \\
+\mathrm{pbf}_{B+1}(1 \mid n)\; (\max(\mathrm{t}_1, \mathrm{T}_{n+1}) - \mathrm{T}_{n+1}) \\
+\mathrm{pbf}_{B+1}(2 \mid n)\; (\max(\mathrm{t}_1, \mathrm{t}_2, \mathrm{T}_{n+1}) - \mathrm{T}_{n+1}) \\
\qquad\qquad\vdots \\
+\mathrm{pbf}_{B+1}(n \mid n)\; (\max(\mathrm{t}_1, \mathrm{t}_2, \ldots, \mathrm{t}_n, \mathrm{T}_{n+1}) - \mathrm{T}_{n+1})
\end{array}\right. \\
&= \sum_{i=0}^{n} \mathrm{pbf}_{B+1}(i \mid n)\; (\max(\mathrm{t}_1, \mathrm{t}_2, \ldots, \mathrm{t}_i, \mathrm{T}_{n+1}) - \mathrm{T}_{n+1}). \\
\mathrm{FC}R_n^{\mathrm{Ord},\,j} &= \sum_{i=0}^{n} \mathrm{pbf}_j(i \mid n)\; (\max(\mathrm{t}_1, \mathrm{t}_2, \ldots, \mathrm{t}_i, \mathrm{T}_{n+1}) - \mathrm{T}_{n+1}).
\end{aligned}$$

As in the previous batching scenarios, we derive the expected resequencing delay for a message in the backward flush batching scenario (BFR) conditioned on the number of busy links in the system. We then remove the conditioning. That is,

$$\mathrm{E}[\mathrm{BF}R] = \sum_{n=0}^{N-1} pb(n)\; \mathrm{E}[\mathrm{BF}R_n],$$

$$= \sum_{n=0}^{N-1} pb(n)\, \mathrm{E}\left[\left(\frac{1}{B+1}\right) \mathrm{FC}R_n^{\mathrm{BF},\,B+1} + \left(\frac{1}{B+1}\right) \sum_{j=1}^{B} \mathrm{FC}R_n^{\mathrm{Ord},\,j}\right].$$

We substitute expressions for $\mathrm{FC}R_n^{\mathrm{BF},\,B+1}$ and $\mathrm{FC}R_n^{\mathrm{Ord},\,j}$ and then simplify the result with solutions previously displayed.

$$\begin{aligned}
\mathrm{E}[\mathrm{BF}R] &= \left(\frac{1}{B+1}\right)\left[\sum_{n=0}^{N-1} pb(n) \sum_{i=0}^{n} \mathrm{pbf}_{B+1}(i \mid n) \mathrm{E}[\max(\mathrm{t}_1, \mathrm{t}_2, \ldots, \mathrm{t}_i, \mathrm{T}_{n+1}) - \mathrm{T}_{n+1}] \right. \\
&\quad \left. + \sum_{n=0}^{N-1} pb(n) \sum_{j=1}^{B} \sum_{i=0}^{n} \mathrm{pbf}_j(i \mid n) \mathrm{E}[\max(\mathrm{t}_1, \mathrm{t}_2, \ldots, \mathrm{t}_i, \mathrm{T}_{n+1}) - \mathrm{T}_{n+1}]\right], \\
&= \left(\frac{1}{B+1}\right)\left[\sum_{n=0}^{N-1} pb(n) \sum_{i=0}^{n} \mathrm{pbf}_{B+1}(i \mid n) \left(\frac{1}{\mu} \sum_{k=2}^{i+1} \frac{1}{k}\right) \right. \\
&\quad \left. + \sum_{j=1}^{B} \sum_{n=0}^{N-1} pb(n) \sum_{i=0}^{n} \mathrm{pbf}_j(i \mid n) \left(\frac{1}{\mu} \sum_{k=2}^{i+1} \frac{1}{k}\right)\right]. \qquad (2.20)
\end{aligned}$$

We combine this expected resequencing delay with the waiting and transmission expected delays to obtain the total delay a message may expect in this batching scenario:

$$\begin{aligned}
\mathrm{E}[\mathrm{BF}D] &= \frac{p_0}{\lambda N!} \frac{\rho\,(N\rho)^N}{(1-\rho)^2} + \frac{1}{\mu} + \left(\frac{1}{B+1}\right)\left[\sum_{n=0}^{N-1} pb(n) \sum_{i=0}^{n} \mathrm{pbf}_{B+1}(i \mid n) \left(\frac{1}{\mu} \sum_{k=2}^{i+1} \frac{1}{k}\right) \right. \\
&\quad \left. + \sum_{j=1}^{B} \sum_{n=0}^{N-1} pb(n) \sum_{i=0}^{n} \mathrm{pbf}_j(i \mid n) \left(\frac{1}{\mu} \sum_{k=2}^{i+1} \frac{1}{k}\right)\right]. \qquad (2.21)
\end{aligned}$$

Lastly, we consider $\mathrm{E}[\mathrm{BF}R]$ as B, ρ, and N individually increase. Equation (2.20) shows that $\mathrm{E}[\mathrm{BF}R]$ monotonically decreases when B increases and all other variables are fixed. When ρ increases, the expected resequencing delay in the backward flush batching scenario monotonically increases as well. The argument in support of this fact follows the corresponding argument in the two-way flush batching scenario and is, therefore, omitted. As before, the expected resequencing delay is bounded at $\rho = 1.0$. Finally, consider $\mathrm{E}[\mathrm{BF}R]$ when N is increased. From equation (2.20) we realize that the expected value monotonically increases as N increases. Again, this argument is omitted since it is similar to the one presented in Section 2.10.3. E[BFR] converges to the expected resequencing delay in a system that transmits backward flush batches and has an infinite number of links.

$$\begin{aligned} \mathrm{E}[\mathrm{BF}R_\infty] &= \sum_{n=0}^{\infty} p_n \, \mathrm{E}[\mathrm{BF}R_n], \\ &= \left(\frac{1}{B+1}\right)\left[\sum_{n=1}^{\infty} p_n \sum_{i=0}^{n} \mathrm{pbf}_{B+1}(i \mid n)\left(\frac{1}{\mu}\sum_{k=2}^{i+1}\frac{1}{k}\right)\right. \\ &\qquad \left. + \sum_{j=1}^{B}\sum_{n=1}^{\infty} p_n \sum_{i=0}^{n} \mathrm{pbf}_j(i \mid n)\left(\frac{1}{\mu}\sum_{k=2}^{i+1}\frac{1}{k}\right)\right]. \end{aligned} \tag{2.22}$$

All of the derivations in this section require $\mathrm{pbf}_c(i \mid n)$. We know that

$$\sum_{i=0}^{n} \mathrm{pbf}_c(i \mid n) = 1$$

and

$$\mathrm{pbf}_c(i \mid n) = \frac{\mathrm{pbf}_c(i \wedge n)}{\sum_{i=0}^{n} \mathrm{pbf}_c(i \wedge n)} \tag{2.23}$$

where

$$\mathrm{pbf}_c(i \wedge n) = P\{i \text{ backward flush messages in transit } \wedge n \text{ busy links}\}.$$

Without re-iterating the discussion in Section 2.10.3, we approximate $\mathrm{pbf}_c(i \wedge n)$ with $\mathrm{pbf}^*_c(i \wedge n)$. That is, in the approximation, we do not acknowledge queueing delay in a system with a finite number of links and we assume the individual components within the probability, e.g., $T < A, T < 2A, \ldots$ within $P\{T < A \wedge T < 2A \wedge \ldots\}$, are independent. Let us consider specific examples.

Suppose a backward flush message is tagged and ready for transmission. Then

$$\begin{aligned} \mathrm{pbf}^*_{B+1}(0 \wedge 0) &= P\{T < A\}P\{T < 2A\}P\{T < 3A\}\ldots, \\ &= (1-\alpha)(1-\alpha^2)(1-\alpha^3)\ldots. \end{aligned}$$

If the tagged message is an ordinary message instead of a backward flush message, then the probability is equivalent; i.e., given c, $\mathrm{pbf}_c^*(0 \wedge 0) = \mathrm{pbf}_{B+1}^*(0 \wedge 0)$.

For a second example, consider the transmission of a tagged message with c message transmissions, $c = B + 1$ if the tagged message is a backward flush, since the transmission of the previous backward flush message.

$$\begin{aligned}
\mathrm{pbf}_c^*(0 \wedge 1) \;=\; & \Big[(1-\alpha^c)(1-\alpha^{2c})(1-\alpha^{3c})(1-\alpha^{4c})\ldots\Big] \\
& \times \Big[\alpha\,(1-\alpha^2)(1-\alpha^3)\ldots(1-\alpha^{c-1})(1-\alpha^{c+1})(1-\alpha^{c+2})\ldots \\
& \quad + (1-\alpha)\,\alpha^2\,(1-\alpha^3)\ldots(1-\alpha^{c-1})(1-\alpha^{c+1})(1-\alpha^{c+2})\ldots \\
& \quad \vdots \quad \Big].
\end{aligned}$$

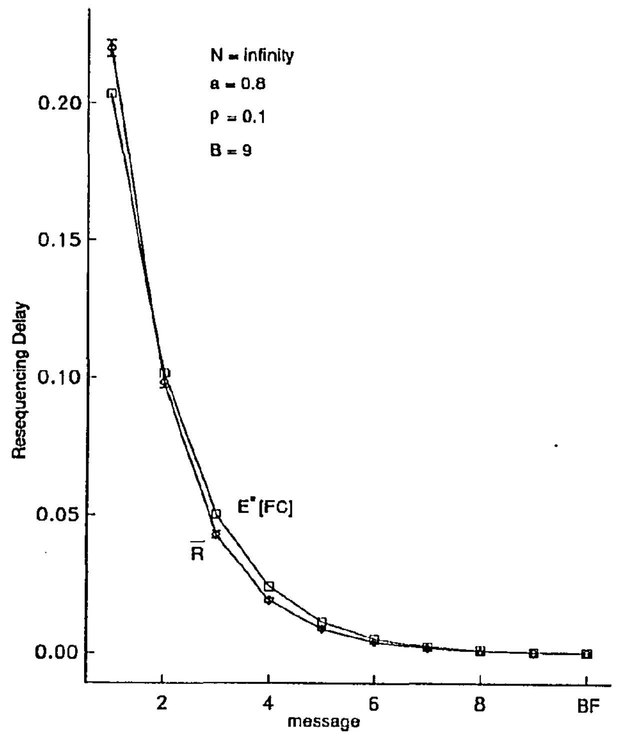

Figure 2.33: Analysis vs. Simulation Results: $M/M/\infty$

$$
\begin{aligned}
\mathrm{pbf}^*_c(1 \wedge 1) \;=\; & \Big[\alpha^c\,(1-\alpha^{2c})(1-\alpha^{3c})(1-\alpha^{4c})\ldots \\
& + (1-\alpha^c)\,\alpha^{2c}\,(1-\alpha^{3c})(1-\alpha^{4c})\ldots \\
& \qquad \vdots \quad \Big] \\
& \times \Big[(1-\alpha)(1-\alpha^2)(1-\alpha^3)\ldots(1-\alpha^{c-1})(1-\alpha^{c+1})(1-\alpha^{c+2})\ldots\Big].
\end{aligned}
$$

The above two examples indicate the method to obtain an approximation of $\mathrm{pbf}_c(i \wedge n)$. The approximation $\mathrm{pbf}^*_c(i \wedge n)$ is a summation of probabilities. Each probability is a product of two products. The first product is the probability that i backward flush messages are in transit; the second product is the probability that $n - i$ ordinary messages are in transit. The value c aids us in locating the backward flush messages. The number of possible combinations for both the first and second products is ∞.

In Figure 2.33, we plot the approximate expected resequencing delay for nine ordinary

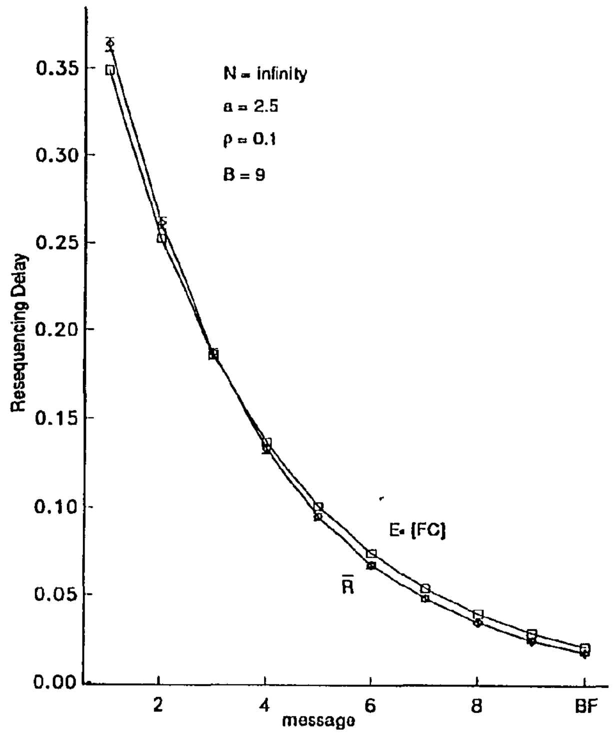

Figure 2.34: Analysis vs. Simulation Results: $M/M/\infty$

messages in a batch, $E^*[FC^{\mathrm{Ord},j}]$, and the approximate expected resequencing delay for the backward flush message when $a = \frac{\lambda}{\mu} = 0.8$. (Recall that a represents the average number of busy links in an $M/M/\infty$ system.) We compare this result to $\bar{R}$ for a simulated 8-link system with $\lambda = 0.8$ and $\mu = 1.0$. As the figure illustrates, the analytical and simulation results validate each other in the backward flush batching scenario as well. The mean resequencing delay from the simulator is between 0.0389 and 0.0403. The result from the analysis is 0.0400.

As in the two-way flush batching scenario, we compare the approximate expected resequencing delay for nine ordinary messages when the average number of busy links is $a = 2.5$ to $\bar{R}$ for a simulated 25-link system with $\lambda = 2.5$ and $\mu = 1.0$. Figure 2.34 plots the comparison of the singular messages. The mean resequencing delay from the simulator is between 0.1213 and 0.1250. The analysis approximates the expected resequencing delay at 0.1241.

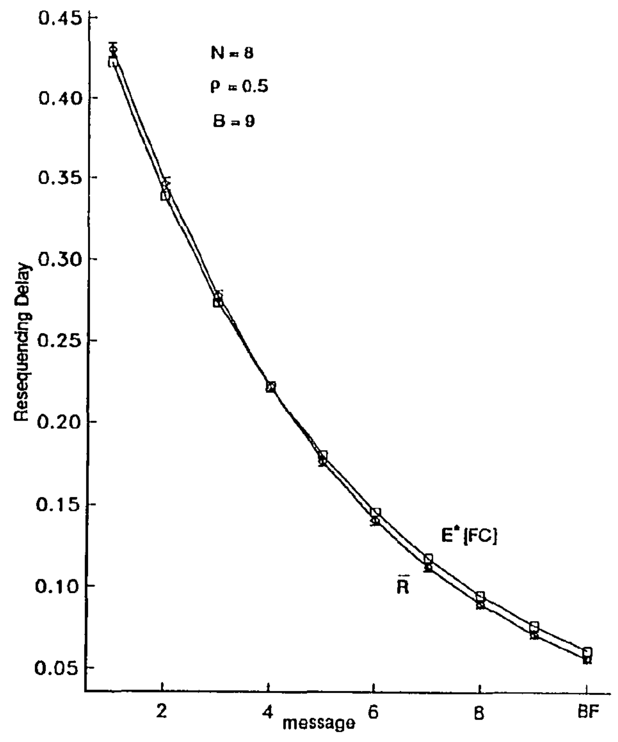

Figure 2.35: Analysis vs. Simulation Results: $M/M/8$

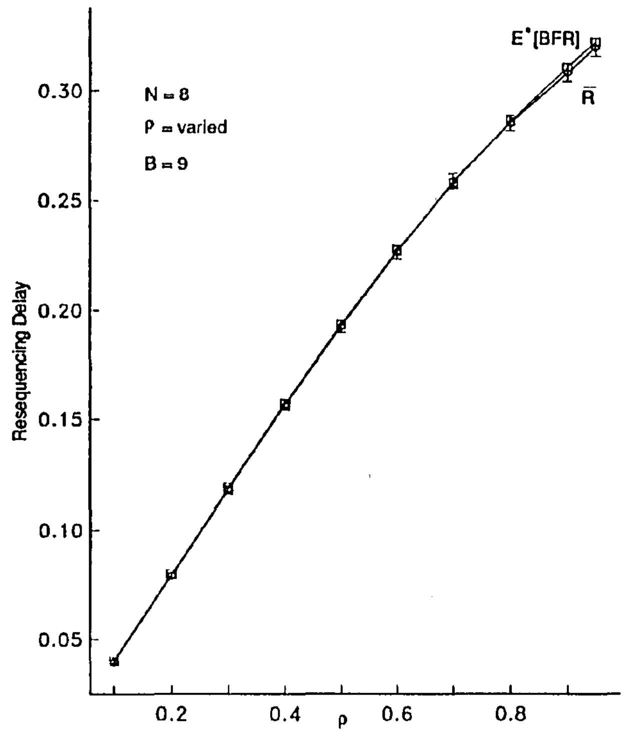

Figure 2.36: Analysis vs. Simulation Results: *M/M/8*

Figure 2.35 plots the approximate expected resequencing delay for nine ordinary messages, $E^*[FC^{Ord, j}]$, and the approximate expected resequencing delay for the backward flush message when $\lambda = 4.0$ and $\mu = 1.0$ in an *M/M/8* system. We compare this result to the corresponding simulation result. In this case, the approximate analysis produces an expected resequencing delay of 0.1932; the confidence interval from the corresponding simulation result is 0.1895 to 0.1949.

Figure 2.36 plots the two expected resequencing delays of the backward flush batching scenario, simulation and approximate analysis, for an 8-link system. Although utilization is varied from no queueing delay to system saturation, the simulation and approximate analysis results continue to validate one another.

'For example' is not proof.
Jewish Proverb

Chapter 3

Verification of a Flush Channel

The previous chapter shows that F-channel communication allows the possibility of higher bandwidth than communication by a virtual circuit. Unfortunately for the user, however, system programming which uses F-channels is more complex than the conventional virtual circuit paradigm. To handle the additional program complexity, we develop an axiomatic verification methodology for F-channel communication.

3.1 Importance of Program Verification

Formal program verification is not heavily used in software development because powerful proof techniques are complex and tedious to apply. Instead, programmers convince themselves of program correctness by executing their program with different input cases. After a wide variety of test inputs leads to the intended results, the programmer has increased confidence that the program is correct. Program testing, in this manner, often accounts for more than half the time spent on the entire programming project [Hoa69]. In the case of distributed systems, arguing correctness via testing is even more problematic. The difficulty originates from the large number of execution interleaving possible in concurrency. To illustrate, there are $n!$ possible execution orders for n concurrent atomic operations in a distributed system; testing for correctness must consider each of these $n!$ possible executions.

In addition to the inadequacies of program testing, there are other advantages to formal verification. One can place greater reliance in a system that has been formally verified. Such reliance is generally impossible if testing is done exclusively. Another benefit is that formal proof techniques uncover invariants of concurrent systems which further increase the understanding of the entire system. In fact, one may postulate that effective testing and formal verification should be viewed as two sides of the same coin [LK91].

3.2 System Communication States

In order to formally discuss distributed programs, a definition of the state of the system is essential [LS84]. The state of a distributed program has three components:

- The *data state* is the mapping between program variables and their values.
- The *control state* is the mapping between the program counters of the programs' processes and operations in the executable code of processes. It tracks the loci of control for the individual processes.
- The *communication state* maps "the network" (hardware and software) onto messages sent, received, and in transit. It allows one to deduce the messages which could possibly be delivered to the destination process.

Implicit variables, variables to which a program makes no explicit assignment, are typically used in operational developments of control and communication state. They encapsulate key aspects of the state of a process, but may not affect the execution of the overall system in any way [OG76]. For example, a process' program counter is most useful in Hoare-style verification proofs [Lam88]; its value is altered as a process executes, but a declaration of, or direct assignment to, a program counter never appears in a program. Schlichting and Schneider [SS84] use several data structures to model the communication state of distributed processes using asynchronous communication. The axiomatic semantics of a message passing construct are expressed in terms of how the implicit variables which model the communication state are affected by execution of the construct.

3.3 Background: Axiomatic Proof Methodology

A distributed program consists of a number of separate processes. These processes share no common memory, and hence, a process may not access another process' variables. We extend the axiomatic techniques for verification of concurrent and distributed programs [LG81, OG76, SS84] that require three steps in the proof of a distributed program: a proof in isolation, a satisfaction proof, and a non-interference proof. In [AFR80], the power of synchronous communication is used to forgo a non-interference proof. Apt avoids this third step by requiring that all assertions only reference local states. In [MC81] and [Sou84], the authors also consider synchronous message passing. These two verification techniques further stress the importance of the proof in isolation by defining invariants that describe process interactions. Since we are concerned with asynchronous communication, we review all three steps in the verification process presented by [LG81, OG76, SS84]. The first step is a *Proof in Isolation* of all processes which comprise the program. This is achieved through a consistent Hoare-style annotation of each process. Let S be an executable statement (atomic operation) in a process. The Hoare triple $\{P\}S\{Q\}$ means that if S is started in a state which satisfies P and S terminates, then Q must hold [Hoa69]. P is the precondition of S, pre(S); Q is its postcondition, post(S). P and Q are also termed *verification assertions.* The relationship between post(S) and pre(S) are defined by the semantics of S. For example, consider the assignment statement,

$$x := f,$$

where x is a variable and f is an expression. Any postcondition of the statement must also be true before the assignment, but with the old value of x. This fact is expressed formally as the

Axiom of Assignment:

$$\{Q^x_f\} \quad x := f \quad \{Q\},$$

where Q^x_f denotes the textual substitution of f for every free occurrence of x in Q [Hoa69].

If each atomic action in a process is annotated with preconditions and postconditions such that its precondition holds when control is immediately before the action and its post-condition holds when control is immediately after the action, then the process is annotated consistently. For every axiomatic proof methodology that follows, we only consider the assertions surrounding communication statements. The remainder of the proof in isolation, verified with traditional Hoare-style annotations, is identical for the different communication paradigms. The *Satisfaction Proof* assures that postconditions of receives are consistent with data transmitted by other processes. This step in the verification of a program is necessary since postconditions of receives cannot be verified in isolation—the postcondition may make unsubstantiated claims about data values being assigned. The *Non-interference Proof* establishes that assertions in one process are not invalidated by actions in another; i.e., actions of one process do not interfere with assertions in any other. Once these three separate proofs are completed, the distributed program is considered formally verified.

3.3.1 Synchronous Communication

Synchronous message operations are defined in Hoare's Communicating Sequential Processes (CSP) [Hoa78]. An interaction between two processes in CSP can be regarded as a distributed assignment statement. One serious flaw, which Hoare admits, is the lack of a proof methodology for verification. Levin and Gries extended CSP to develop these missing proof rules [LG81]. For a set of distinct communicating processes, let us consider the three steps that are involved to prove correctness: a proof in isolation, a satisfaction proof, and a non-interference proof.

The proof in isolation is in the Hoare-style discussed above. To complete the proof, the verifier first annotates each process in isolation with assertions of the form $\{P\}\ S\ \{Q\}$. Using the axioms and inference rules of the proof system, the verifier proves precondition P implies postcondition Q upon termination of each statement. When the statement in a process is a communication with another process, one cannot verify P implies Q. The soundness of the methodology, however, requires that Q be justified. In the proof in isolation, any postcondition is allowed—communication statements never terminate in isolation—therefore, the verifier assumes that Q is correct (a "miraculous postcondition") in isolation. The role of the satisfaction proof is to verify this assumption [LG81].

Suppose P_i and P_j are two processes in a communicating system. In process P_i, the assertions surrounding the transmission statement are

$$\{R\}\ P_j!\text{expr}\ \{T\}.$$

For every matching **receive** statement in P_j,

$$\{P\}\ P_i?\text{var}\ \{Q\},$$

we use the satisfaction rule to verify the postconditions of the communication:

Synchronous Satisfaction Rule:

For every synchronous **receive** statement and every matching **send statement**, verify the following to establish satisfaction:

$$(R \wedge P) \Rightarrow (T \wedge Q)^{\text{var}}_{\text{expr}}.$$

The third and final step for proving the correctness of a distributed program is non-interference. A non-interference proof is required when assertions in one process refer to variables in another process. One must show that for each assertion A, and for every statement, S, *parallel* to A, the execution of S preserves the truth of A. S is considered parallel to A if S is a statement in one process and A is an assertion in another. Assume that we are in a programming context in which only the execution of assignment statements, **send** statements, and **receive** statements alter the data or communication state of a program. The full impact of an assignment statement is given by the assignment axiom. If a **send** and a **receive** are a matching communication pair, their impact upon the data and communication state must be expressed in terms of satisfaction.

Synchronous Non-interference Rule:

For assertion A and parallel assignment, **send** statement, or **receive** statement S, prove

$$\{A \wedge \text{pre}(S)\} \quad S \quad \{A\}.$$

For assertion A and matching parallel P_j!expr and P_i?var statements, prove

$$(A \wedge \text{pre}(P_j!\text{expr}) \wedge \text{pre}(P_i?\text{var})) \Rightarrow A\ ^{\text{var}}_{\text{expr}}.$$

3.3.2 Asynchronous Communication

Schlichting and Schneider extended the proof technique of Levin and Gries from synchronous to asynchronous message passing in [SS84]. Of interest in this work is that they developed proof rules for interprocess communication via unreliable datagrams and reliable virtual circuits. With unreliable datagrams, messages transmitted through the communication channel may be delivered in any order (if delivered at all). In the case of reliable virtual circuits, messages are assured delivery in the same order as they were transmitted. The following two discussions review the proof rules for these two types of asynchronous communication.

Unreliable Datagrams:

Following the formal proof technique of Levin and Gries [LG81], proving the correctness of a system with unreliable datagram communication also requires three steps: a proof in isolation, a satisfaction proof, and a non-interference proof. Schlichting and Schneider employ two implicit variables for each process to model the communication state [SS84]. One variable is the send multiset, σ_D, for process D. A copy of every message transmitted to process D is contained in σ_D. Likewise, the receive multiset, ρ_D, includes a copy of every message received by process D. As messages can only be received if they have been transmitted, the system obeys the following axiom:

Unreliable Datagram Network Axiom:

$$\rho_D \subseteq \sigma_D.$$

The proof in isolation of a distributed system, communicating with unreliable datagrams, must take into account the asynchronous nature of the communication. Execution of a transmission statement in synchronous communication blocks the process until receipt occurs. In the asynchronous situation, the execution of the statement

$$\textbf{send}\ \text{msg}\ \textbf{to}\ D$$

has the semantic impact of adding the message to σ_D ($\sigma_D := \sigma_D \oplus \{\text{msg}\}$). The sender then continues executing. For the proof in isolation, the assertions surrounding **send** statements reflect the fact, using the assignment axiom, that the transmission merely inserts msg into σ_D.

Unreliable Datagram Send Axiom:

$$\{W^{\sigma_D}_{\sigma_D \oplus \{\text{msg}\}}\}\ \textbf{send}\ \text{msg}\ \textbf{to}\ D\ \{W\}.$$

Since a datagram **receive** is synchronous, its postcondition is miraculous.

Unreliable Datagram Receive Axiom:

$$\{P\}\ \textbf{receive}\ \text{msg}\ \{Q\}.$$

Consider the **receive** statement above. By the Unreliable Datagram Network Axiom, a process D cannot execute the **receive** unless there is a message available in the communication channel. Suppose $MTEXT$ is a message that has been transmitted to process D, but has not been received; hence $MTEXT \in (\sigma_D \ominus \rho_D)$ where $\ominus$ is the multiset difference operator. Execution of the **receive** results in the addition of $MTEXT$ to ρ_D and the assignment of $MTEXT$ to msg. It is equivalent to the dual assignment

$$\text{msg}, \rho_D := MTEXT, \rho_D \oplus \{MTEXT\}.$$

Unreliable Datagram Satisfaction Rule:

For every unreliable datagram **receive** statement, verify the following to establish satisfaction:

$$(P \wedge MTEXT \in (\sigma_D \ominus \rho_D)) \Rightarrow Q\ {}^{\text{msg},}_{MTEXT,}{}^{\rho_D}_{\rho_D \oplus \{MTEXT\}}.$$

The last proof rule required for unreliable datagrams is non-interference.

Unreliable Datagram Non-interference Rule:

For assertion A and parallel assignment, **send** statement, or **receive** statement S, prove

$$\{A \wedge \text{pre}(S)\} \quad S \quad \{A\}.$$

For assertion A and parallel **receive** statement S, prove

$$(A \wedge \text{pre}(S) \wedge MTEXT \in (\sigma_D \ominus \rho_D)) \Rightarrow A\ {}^{\text{msg},}_{MTEXT,}{}^{\rho_D}_{\rho_D \oplus \{MTEXT\}}.$$

Virtual Circuits

Virtual circuits are very popular in many communication networks; the communication channel becomes a totally reliable FIFO queue. The proofs concerning distributed systems using virtual circuits require the three customary steps: a proof in isolation, a satisfaction proof, and a non-interference proof. To model the communication state, two implicit variables for each circuit are required. The variable σ_V maintains the messages transmitted on the virtual circuit V, while the variable ρ_V records the messages received from the virtual circuit. Both of these implicit variables are ordered sequences of messages. In [SS84] the following operations on two sequences, C_1 and C_2, are defined:

$C_1 \leq C_2$	is true if C_1 is a prefix of C_2,
$C_1 + \text{val}$	is the sequence obtained by appending val to C_1,
$C_1 - C_2$	is the sequence that results from deleting prefix C_2 from C_1,
$hd(C_1)$	is the first element in C_1.

The Virtual Circuit Network Axiom insists that the virtual circuit be a fully reliable FIFO channel.

Virtual Circuit Network Axiom:

$$\rho_V \leq \sigma_V.$$

We now review the proof methodology for virtual circuit communication. Execution of a statement

$$\textbf{send } \text{msg } \textbf{on } V \quad .$$

is identical to *appending* the message to the implicit variable σ_V ($\sigma_V := \sigma_V + \text{msg}$). The process then continues executing.

Virtual Circuit Send Axiom:

$$\{W^{\sigma_V}_{\sigma_V + \text{msg}}\} \textbf{ send } \text{msg } \textbf{on } V \ \{W\}.$$

Except for the different properties involved in manipulating σ_V, this axiom is identical to the **send** axiom for unreliable datagrams. The **receive** axiom is miraculous:

Virtual Circuit Receive Axiom:

$$\{P\} \text{ **receive** msg **from** } V \ \{Q\}.$$

The satisfaction rule for virtual circuits validates the miraculous postconditions of **receive** statements. If the **receive** statement above is executing and $MTEXT = hd(\sigma_V - \rho_V)$, then $MTEXT$ will be assigned to msg and will be appended to the sequence ρ_V. It is equivalent to the dual assignment

$$\text{msg}, \rho_V := MTEXT, \rho_V + MTEXT.$$

Virtual Circuit Satisfaction Rule:

For every virtual circuit **receive** statement, verify the following to establish satisfaction:

$$(P \wedge (\sigma_V - \rho_V) \neq \phi \wedge MTEXT = hd(\sigma_V - \rho_V)) \Rightarrow Q \ {}^{\text{msg},}_{MTEXT,}{}^{\rho_V}_{\rho_V + MTEXT}.$$

The following non-interference rule proves that the assertions in the proof are *globally* true.

Virtual Circuit Non-interference Rule:

For assertion A and parallel assignment, **send** statement, or **receive** statement S, prove

$$\{A \wedge \text{pre}(S)\} \quad S \quad \{A\}.$$

For assertion A and parallel **receive** statement S, prove

$$(A \wedge \text{pre}(S) \wedge (\sigma_V - \rho_V) \neq \phi \wedge MTEXT = hd(\sigma_V - \rho_V)) \Rightarrow A \ {}^{\text{msg},}_{MTEXT,}{}^{\rho_V}_{\rho_V + MTEXT}.$$

3.4 An Axiomatic Proof Methodology for Flush Channels

Communication with F-channels is different from the previously discussed asynchronous communication paradigms due to the dynamic nature of the delivery order requirements. We can no longer model the delivery restrictions of the communication state as a static network axiom. Instead, we must *construct* the delivery order within the send axioms. We model the communication state of F-channel F as follows. Let σ_F denote the *send multiset* for F, and let ρ_F denote the *receive multiset* for F. We define $\prec\!+_F$ on the multiset σ_F,

$$\prec\!+_F \subseteq \sigma_F \times \sigma_F,$$

such that for $m, m' \in \sigma_F$, $m \prec\!+_F m'$ if and only if m cannot be delivered after m'. The $\prec\!+_F$ relation is an irreflexive partial order constructed by the delivery order semantics of the messages transmitted over F. We find it convenient to define the *covering relation* of $\prec\!+_F$,

$$\prec_F \subseteq \sigma_F \times \sigma_F,$$

as the smallest relation such that its transitive and irreflexive closure is $\prec\!+_F$. In other words, if $m, m' \in \sigma_F$ and $m \prec\!+_F m'$, $m \prec_F m'$ if and only if there is no $m'' \in \sigma_F$ such that $m \prec\!+_F m'' \prec\!+_F m'$. As each message is transmitted on F, $\prec_F$ is modified to reflect delivery order requirements for the message relative to those messages transmitted previously. As messages are transmitted, $\prec_F$ is constructed incrementally. $\prec\!+_F$ is obtained by closure.

A system using an F-channel has two properties concerning the defined implicit variables. Like the previous proof methodologies, a message cannot be received if it has not been transmitted. $\prec\!+_F$ specifies the required delivery order of each message in σ_F.

Flush Channel Network Axiom: For F-channel F, the following two properties must hold:

> *En Route Property:* $\rho_F \subseteq \sigma_F$
>
> *Order Property:* For $m, m' \in \sigma_F$, $m' \in \rho_F \Rightarrow (\forall m : m \prec\!\!\!+_F m' :: m \in \rho_F)$.

Given the network axiom for F-channels, we may proceed with the proof methodology. The next three sections give axiomatic proof rules for F-channels following the traditional proof technique: a proof in isolation, a satisfaction proof, and a non-interference proof. The rules are results developed from extending the methodologies presented in Section 3.3. In fact, in Section 3.10, we show that the following axiomatic proof methodology for F-channels is a true generalization of the verification process for communication with reliable datagrams and virtual circuits.

3.4.1 Proof in Isolation

The semantics of an F-channel **send** are presented for four cases, each case corresponds to the type of the message being sent. Let m denote the composite <type, data>, the message which is transmitted, in all four of the transmission axioms. To aid in the construction of $\prec\!\!\!+_F$, the partial order specifying delivery constraints, two additional implicit variables are necessary. The *backward flush point*, τ_F, is a set which contains the last two-way or backward flush transmitted on F. As defined in Section 2.1, any message transmitted after a backward flush point must be delivered after the message defining this point. The *free set*, Ω_F, is a multiset of messages transmitted on F which have no successor in $\prec\!\!\!+_F$. At the transmission of a message that flushes the channel in a forward direction (forward flush and two-way flush messages), all messages in the free set and their predecessors must be delivered before the message being transmitted. Both τ_F and Ω_F are initially empty. During the course of message transmission, Ω_F may become arbitrarily large; τ_F will be at most a singleton. The following four axioms are necessary for a proof in isolation.

Ordinary Message Transmission Consider the execution of

S_1: **send** <**Ord**, data> **on** F.

When an ordinary message is transmitted, it is added to σ_F. The notation $\sigma_F := \sigma_F \oplus \{\mathrm{m}\}$ denotes that m is added to the multiset σ_F. Additionally, an ordinary message must be guaranteed delivery after the backward flush point. Thus, the partial order is expanded to ensure that $\tau_F \prec\!\!\!+_F \mathrm{m}$. The newly sent message will also be added to the set of free messages—nothing follows it in the partial order (yet). Furthermore, the new message may remove the current backward flush point, τ_F, from the free set. τ_F now has at least one successor in the partial order. Operationally speaking then, the net impact of the transmission of an ordinary message, m, is simply the multiple assignment:

$$\sigma_F, \prec_F, \Omega_F := \sigma_F \oplus \{\mathrm{m}\}, \prec_F \oplus A(\mathrm{m}), \Omega_F \oplus \{\mathrm{m}\} \ominus B(\mathrm{m}),$$

where $A(\mathrm{m}) = \{(x, \mathrm{m}) \mid x \in \tau_F\}$ and $B(\mathrm{m}) = \{x \mid x \in \tau_F \wedge x \in \Omega_F\}$. As a notational contraction in the assignment to $\prec_F$, $A(\mathrm{m})$ represents the potential additional element of the partial order which results when m is linked to a non-empty τ_F. Likewise, $B(\mathrm{m})$ represents the potential deletion of τ_F from the free set.

If $W = \mathrm{post}(S_1)$, then the assignment axiom allows us to deduce $\mathrm{pre}(S_1)$. Execution of S_1 is equivalent to the three assignments above; namely

Ordinary Message Send Axiom:

$$\{W^{\sigma_F,\ \prec_F,\ \Omega_F}_{\sigma_F \oplus \{\mathrm{m}\},\ \prec_F \oplus A(\mathrm{m}),\ \Omega_F \oplus \{\mathrm{m}\} \ominus B(\mathrm{m})}\} \quad S_1 : \textbf{send} <\textbf{Ord}, \mathrm{data}> \textbf{on } F\ \{W\}.$$

Two-way Flush Transmission When we transmit a two-way flush, m,

S_2: **send** <**2F**, data> **on** F, we impose substantial delivery ordering restrictions on the F-channel. First, the two-way flush must be delivered after every element of the free set (as defined at the instant of m's transmission). Second, m's delivery must precede the delivery of every message transmitted after m. In operational terms,

$$\sigma_F, \prec_F, \Omega_F, \tau_F := \sigma_F \oplus \{\mathrm{m}\}, \prec_F \oplus C(\mathrm{m}), \{\mathrm{m}\}, \{\mathrm{m}\}$$

where $C(\mathrm{m}) = \{(x, \mathrm{m}) \mid x \in \Omega_F\}$. The addition of $C(\mathrm{m})$ to the partial order effectively means that any message which was in the free set just prior to S_2 cannot be delivered after m. m then becomes the new free set (it has no successors yet), and it becomes the new backward flush point (it must be delivered before every subsequently transmitted message). In sum,

Two-way Flush Send Axiom:

$$\{W^{\sigma_F,\ \prec_F,\ \Omega_F,\ \tau_F}_{\sigma_F \oplus \{\mathrm{m}\}, \prec_F \oplus C(\mathrm{m}), \{\mathrm{m}\}, \{\mathrm{m}\}}\} \quad S_2 : \textbf{send} <\textbf{2F}, \text{data}> \textbf{on } F\ \{W\}.$$

Forward Flush Transmission A forward flush, m, is transmitted by

S_3: **send** <**FF**, data> **on** F.

Like a two-way flush, the delivery of a forward flush must be guaranteed after all messages in the free set; m becomes the only member of the free set. Unlike a two-way flush, however, a forward flush does not become the backward flush point—messages transmitted after m may be delivered before m. Transmission of a forward flush makes the following implicit assignments:

$$\sigma_F, \prec_F, \Omega_F := \sigma_F \oplus \{\mathrm{m}\}, \prec_F \oplus C(\mathrm{m}), \{\mathrm{m}\}.$$

The set $C(\mathrm{m})$ is as previously defined in the discussion of the two-way flush.

Forward Flush Send Axiom:

$$\{W^{\sigma_F,\ \prec_F,\ \Omega_F}_{\sigma_F\oplus\{\mathrm{m}\},\prec_F\oplus C(\mathrm{m}),\{\mathrm{m}\}}\}\quad S_3 : \textbf{send} <\!\mathbf{FF}, \text{data}\!> \textbf{on } F\ \{W\}.$$

Backward Flush Transmission We transmit a backward flush message, m, with the statement

$$S_4\text{: } \textbf{send} <\!\mathbf{BF}, \text{data}\!> \textbf{on } F.$$

Some messages transmitted before m may be delivered after m (elements of the free set), therefore, m joins the free set. The addition of $\tau_F \prec\!+_F$ m to the partial order also (possibly) removes τ_F from the free set. Lastly, m becomes the new backward flush point—no message transmitted after m may be delivered before it. In operational terms, this means

$$\sigma_F, \prec_F, \Omega_F, \tau_F := \sigma_F \oplus \{\mathrm{m}\}, \prec_F \oplus A(\mathrm{m}), \Omega_F \oplus \{\mathrm{m}\} \ominus B(\mathrm{m}), \{\mathrm{m}\}.$$

The sets $A(\mathrm{m})$ and $B(\mathrm{m})$ are as previously defined for ordinary message transmission.

Backward Flush Send Axiom:

$$\{W^{\sigma_F,\ \prec_F,\ \Omega_F,\ \tau_F}_{\sigma_F\oplus\{\mathrm{m}\},\prec_F\oplus A(\mathrm{m}),\Omega_F\oplus\{\mathrm{m}\}\ominus B(\mathrm{m}),\{\mathrm{m}\}}\}\quad S_4 : \textbf{send} <\!\mathbf{BF}, \text{data}\!> \textbf{on } F\ \{W\}.$$

Message Reception The statement

$$R : \textbf{receive} <\!\text{mtype}, \text{mdata}\!> \textbf{from } F$$

is synchronous; termination of this statement is in no way dependent upon the action of this statement. post(R) is thus allowed to be miraculous. Hence,

Flush Channel Receive Axiom:

$$\{P\} \quad R : \textbf{receive} <\text{mtype}, \text{mdata}> \textbf{from } F \ \{Q\}.$$

Of course, in a satisfaction proof, Q must be justified relative to P and the message received (see Section 3.4.2).

The preceding discussion illustrates how the delivery order covering relation, $\prec_F$, is built as messages are transmitted on the F-channel. $\prec_F$ is extended to $\prec^+_F$ by closure. In order for this construction to be meaningful, we must establish that the delivery order restrictions defined inherently in the F-channel are exactly represented in the structure of $\prec^+_F$. That is, if message m cannot be received after m', then the constructed partial order must reflect this fact. Moreover, we require proof that the addition of $m \prec^+_F m'$ to the partial order implies m cannot be received after m'.

Theorem 10 *Let m and m' be messages transmitted on F-channel F. Then $m \lhd^+_F m'$ if and only if $m \prec^+_F m'$.*

In order to establish this result, several structural properties of $\prec^+_F$ must first be presented. We find it convenient to exploit the graphical representation of the relation in making some of our arguments. The directed graph of a binary relation $\mathcal{R}$ on set S is $G(\mathcal{R}) = \{S, \mathcal{R}\}$. That is, $G(\mathcal{R})$ is a graph on the set of nodes S, where edge (s_1, s_2) is in edge set E if and only if $(s_1, s_2) \in \mathcal{R}$. We switch between the graphical interpretation and the algebraic interpretation as best suits the argument.

Lemma 8 *The binary relation $\prec^+_F$ is an irreflexive partial order.*

Proof: The principal fact necessary for this proof is that on transmission of message m, no element of the form (m, m'), where m' is an *old* element of σ_F, is ever added to $\prec_F$. In graphical terms, this implies that $G(\prec_F)$ is acyclic. Edge (m, m') is in $G(\prec\!\!\prec_F)$ only if there is a directed path of one or more edges from m to m' in $G(\prec_F)$.

The rules for the construction of $\prec_F$ produce an irreflexive, non-transitive, and antisymmetric relation. The transitive and irreflexive closure of $\prec_F$ preserves the irreflexive and antisymmetric properties. $\prec\!\!\prec_F$ is made transitive by the closure operator applied to $\prec_F$. ∎

Recall the relation,

$$\sqsubset_F \subseteq \sigma_F \times \sigma_F.$$

For $x, y \in \sigma_F$, $x \sqsubset_F y$ if and only if x is transmitted before y over F-channel F.

Lemma 9 *Let m be a two-way flush or a forward flush sent on F-channel F. $m' \prec\!\!\prec_F m$ if and only if $m' \sqsubset_F m$.*

Proof: [If] By the axioms which define the semantics of the transmission of a forward or two-way flush, m, $\prec_F := \prec_F \oplus C(m)$ where $C(m) = \{(x, m) \mid x \in \Omega_F\}$. If $m' \in \Omega_F$ when m is transmitted, then $m' \prec_F m$, and clearly $m' \prec\!\!\prec_F m$. Suppose, on the other hand, that $m' \notin \Omega_F$ at the time of the transmission of m. Then there must exist some $m'' \in \Omega_F$ such that $m' \prec\!\!\prec_F m''$; otherwise, m' would be a free element. Then by the appropriate send axiom and closure, after m is transmitted, $m' \prec\!\!\prec_F m$.

[Only If] This follows directly from the construction of $\prec_F$. If $m' \prec\!\!\prec_F m$, then by Lemma 8 m' must have been in $G(\prec_F)$ at the time m was added. ∎

Lemma 9 gives us half of the proof of Theorem 10. We know that if m is a two-way or forward flush, all messages sent before m must be received before m. The lemma shows that $\prec\!\!\prec_F$ reflects this fact exactly. When we consider ordinary messages and backward flushes, the situation is a bit more complex.

Lemma 10 *If m is an ordinary message or a backward flush, and if $m' \prec_F m$, then m' must be a two-way flush or a backward flush, and there is no message $m'' \neq m'$ for which $m'' \prec_F m$.*

Proof: This proof is similar to that in Lemma 2. We, therefore, do not replicate it here. Replace all references to $\lhd_F$ in Lemma 2 with $\prec_F$. ∎

Recall the BFP-chain defined in Section 2.2. In this situation, however, we define the chain using the covering relation instead of the immediate predecessor relation:

$$\text{chain}(m) = \{m_k, m_{k-1}, \ldots, m_1\},$$

where

$$m_k \prec_F m_{k-1} \prec_F \ldots \prec_F m_1 \prec_F m.$$

In the same vein, define $Pred(m) = \{x \mid x \prec\!\!+_F m\}$.

Lemma 11 *If m is an ordinary message or a backward flush, then $m' \in Pred(m)$ if and only if $m' \in \text{chain}(m) \oplus Pred(Head(\text{chain}(m)))$.*

Proof: This proof replicates that in Lemma 3. Replace all references to $\lhd\!+_F$ and $\lhd_F$ in Lemma 3 with $\prec\!\!+_F$ and $\prec_F$ respectively. ∎

Finally, we may establish the truth of Theorem 10 through Lemma 9 and Lemma 11. The former establishes that all messages sent before a two-way or forward flush must be received before the flush is received, and $\prec\!\!+_F$ reflects that fact. The latter precisely describes the predecessor set of an ordinary message or a backward flush. The set consists of elements that must be received before the message that the set defines. Lemma 11 illustrates that this predecessor set is represented directly in $\prec\!\!+_F$. In summary, the partial order, as we construct it in the operational semantics of the F-channel **send** primitives, represents the receipt-order restrictions exactly.

Figure 3.1 illustrates the same sample of immediate predecessors as Figure 1.3. The operational construction of the covering relation may be understood with this graph. We

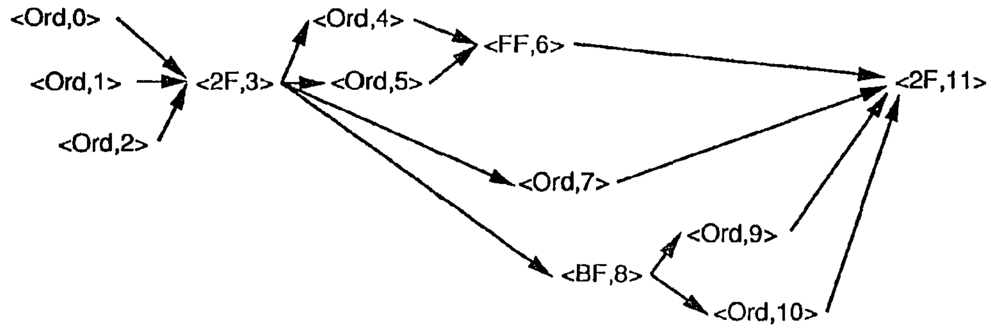

Figure 3.1: A Sample Covering Relation

focus on four messages in order to amplify the rather dry development above.

<**2F**, 3> At the time of transmission of this message, there is no backward flush point and the ordinary messages numbered zero through two are in the free set. All elements of the free set are made predecessors of <**2F**, 3> when it is transmitted. This two-way flush message also becomes the backward flush point—all messages transmitted after it are its successors in $\prec^{+}_{F}$.

<**FF**, 6> When this forward flush is transmitted, the free set consists of the ordinary messages <**Ord**, 4> and <**Ord**, 5>. Their delivery must precede the delivery of <**FF**, 6>, and hence they precede it in $\prec_F$ (and, by closure, in $\prec^{+}_{F}$). It is important to realize that, unlike a two-way flush, a forward flush does not become the new backward flush point. Some messages transmitted after a forward flush may be delivered before it. When <**FF**, 6> is transmitted, it becomes the sole member of the free set. <**2F**, 3> remains the backward flush point.

<BF,8> The delivery of **<BF,8>** may be done in any order relative to **<FF,6>** and **<Ord,7>** as **<BF,8>** joins the free set which includes these messages. The delivery of **<BF,8>** must follow that of the backward flush point at the time of its transmission, **<2F,3>**. Since **<BF,8>** becomes the new backward flush point, its delivery will precede the delivery of every message transmitted after it.

<Ord,9> This message joins the free set, while it removes **<BF,8>**, at the time of its transmission. After the transmission of the message, the free set contains **<FF,6>**, **<Ord,7>**, and **<Ord,9>**. It is also linked into the delivery order so that its delivery succeeds the backward flush point, **<BF,8>**.

3.4.2 Satisfaction

Secure in the fact that the proof in isolation has led to the description of a partial order which is faithful to F-channel semantics, the role of the satisfaction proof is the resolution of the miracle in the F-channel receive axiom.

Consider a specific **receive**, as annotated for the proof in isolation:

$$\{P\}\ R: \textbf{receive}\ \text{<mtype, mdata>}\ \textbf{from}\ F\ \{Q\}.$$

Let *MTEXT* be a message which is eligible for receipt. By the en-route property, it must have been transmitted on the F-channel, but it cannot have been received; i.e., $MTEXT \in \sigma_F \ominus \rho_F$. Its receipt must also be consistent with the order property as specified by $\prec\!\!+_F$. More precisely,

$$\forall m : m \in \sigma_F \wedge m \prec\!\!+_F MTEXT :: m \in \rho_F.$$

If *MTEXT* meets these two requirements, then the **receive** effectively behaves as the dual assignment:

$$\begin{aligned}\text{<mtype, mdata>} &:= MTEXT, \\ \rho_F &:= \rho_F \oplus \{MTEXT\}.\end{aligned}$$

In order to establish $Q = \text{post}(R)$, the above **receive** should be executed in a state which is the weakest precondition[1] [Dij76] of the dual assignment with respect to Q, wp("<mtype, mdata>, $\rho_F := MTEXT, \rho_F \oplus \{MTEXT\}$", Q). Since this statement is simply an assignment, we know that

$$\text{wp(“<mtype, mdata>}, \rho_F := MTEXT, \rho_F \oplus \{MTEXT\}\text{”}, Q) = Q^{\text{<mtype,mdata>}, \rho_F}_{MTEXT, \quad \rho_F \oplus \{MTEXT\}}.$$

In order to verify satisfaction for the receive, we use the precondition and the F-channel network axiom to establish the weakest precondition.

Flush Channel Satisfaction Rule:

For every F-channel receive

$$\{P\} \quad R: \textbf{receive} \text{ <mtype, mdata> } \textbf{from } F \ \{Q\},$$

verify the following to establish satisfaction:

$$P \wedge (MTEXT \in \sigma_F \ominus \rho_F) \wedge (\forall m : m \in \sigma_F \wedge m \prec\!\!\!+_F MTEXT :: m \in \rho_F) \Rightarrow Q^{\text{<mtype,mdata>}, \rho_F}_{MTEXT, \quad \rho_F \oplus \{MTEXT\}}$$

An important part of the proof in isolation of the sender is the establishment of invariants which describe the structure of $\prec\!\!\!+_F$. The structural knowledge is necessary in order to exploit the F-channel network axiom in the satisfaction proof.

[1]The weakest precondition of action S with respect to predicate A, denoted wp(S, A), is the set of all states such that execution of S in any one such state will terminate with A true. If S is the assignment "$x := e$", then, by the assignment axiom, wp("$x := e$", A) is simply A^x_e.

3.4.3 Non-interference

The non-interference rules for parallel assertion and statements is identical to previous proof methodologies. For a **receive** statement, however, we can use both the en route property and the order property to establish the implication.

Flush Channel Non-interference Rule:

For assertion A and parallel assignment, **send** statement, or **receive** statement S, prove

$$\{A \wedge \text{pre}(S)\} \quad S \quad \{A\}.$$

For assertion A and parallel **receive** statement S, prove

$$A \wedge \text{pre}(S) \wedge (MTEXT \in \sigma_F \ominus \rho_F) \wedge (\forall m : m \in \sigma_F \wedge m \prec\!\!+_F MTEXT :: m \in \rho_F)$$
$$\Rightarrow A^{<\text{mtype,mdata}>,\ \rho_F}_{MTEXT,\ \rho_F \oplus \{MTEXT\}}.$$

The preceding development is best justified and appreciated by seeing the methodology applied. In the following section, we apply the methodology to a distributed application that uses all four of the flush message types [CK91]. The example illustrates the tedium that is necessary to correctly verify the application program. In Section 2.8.2, we discussed batching ordinary messages with a flush message type. As the delivery order of each batching example is less complex than a delivery order that uses all four message types, we expect the hardship of the verification process to decrease as well. In Section 3.6, we validate this expectation.

3.5 Verification of a Flush Application

In this section, we apply the axiomatic proof methodology for F-channels to a distributed application that uses all four of the flush message types. In the example, the producer process transmits two arrays (of unknown size initially) to a consumer process. The consumer sums the elements of the array after all the elements have been delivered. Messages are of the form $<type, arnum, index, value>$ where *type* is the message type, *arnum* is the number of the array to which the message belongs, *index* is the index number within the array, and *value* is the value of arnum[index]. If less than four entries are required for a message, then the extra entries are transmitted as zero. A two-way flush message is transmitted to begin the application and denote the end of any previous applications; $(\sigma_F \ominus \rho_F) = \emptyset$ at the delivery of this message. To simplify the example, we assume σ_F and ρ_F are empty before the transmission of the initial two-way flush transmission. Backward flush messages are used to transmit the size of each array to the consumer. The program uses ordinary messages to transmit the elements of the array; these elements cannot be delivered until the size has been delivered. Lastly, forward flush messages denote the end of the array's transmission. At this time, the consumer can sum the array. In the example, the consumer's auxiliary variable X is an array of five sets, initially empty, that contain the messages delivered. Figure 3.2 illustrates the covering relation of this application example.

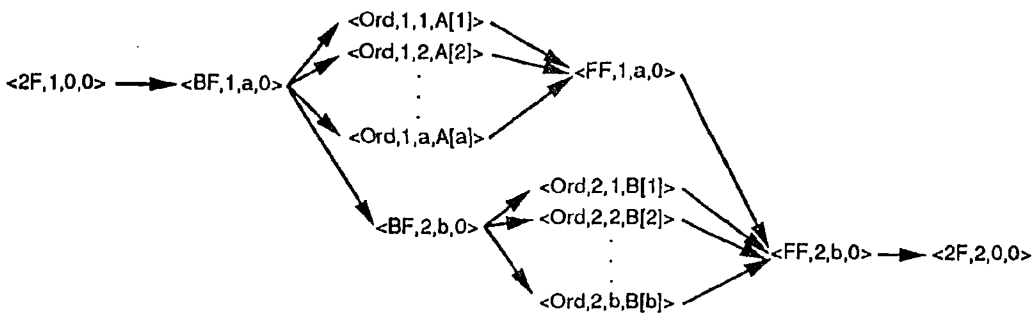

Figure 3.2: The Covering Relation

```
PROD:: var A      : array 1..M of integer;
           B      : array 1..N of integer;
           a,b    : integer;
           i      : integer;

                  ... PREVIOUS APPLICATIONS ...
       s1: send (2F, 1, 0, 0) on F;             *new application message*
       find(a);                                 *find size of first array*
       s2: send (BF, 1, a, 0) on F;             *send size of first array*
       i := 0;
       while i < a do
              i := i + 1;
              s3: send (Ord, 1, i, A[i]) on F;
       od;
       s4: send (FF, 1, a, 0) on F;             *ok to sum first array*
       find(b);                                 *find size of second array*
       s5: send (BF, 2, b, 0) on F;             * send size of second array*
       i := 0;
       while i < b do
              i := i + 1;
              s6: send (Ord, 2, i, B[i]) on F;
       od;
       s7: send (FF, 2, b, 0) on F;             *ok to sum second array*
       s8: send (2F, 2, 0, 0) on F;             *new application message*
                  ... FOLLOWING APPLICATIONS ...


CONS:: var C               : array 1..M of integer;
           D               : array 1..N of integer;
           c,d             : integer;
           mtype           : { 2F, BF, Ord, FF};
           marnum          : integer;
           mindex, mvalue  : integer;
           j               : integer;
           done            : boolean;
           sumC, sumD      : integer;

                           ... PREVIOUS APPLICATIONS ...
      C, D, sumC, sumD, done, X := Φ, Φ, 0, 0, false, Φ;
      while not done do
             r1: receive (mtype, marnum, mindex, mvalue) from F;
             case mtype of
                   2F:  X[1] := X[1] ∪ {marnum};
                        if marnum = 2 then
                          done := true;
```

```
                fi;
          BF:   X[2] := X[2] ∪ {marnum};
                if marnum = 1 then
                   c := mindex
                else
                   d := mindex;
                fi;
          Ord:  if marnum = 1 then
                   C[mindex], X[3] := mvalue, X[3] ∪ {mindex}
                else
                   D[mindex], X[4] := mvalue, X[4] ∪ {mindex};
                fi;
          FF:   j, X[5] := 0, X[5] ∪ {marnum};
                if marnum = 1 then
                   while j < mindex do
                         j := j + 1;
                         sumC := sumC + C[j];
                   od
                else
                   while j < mindex do
                         j := j + 1;
                         sumD := sumD + D[j];
                   od;
                fi;
od;
          ... FOLLOWING APPLICATIONS ...
```

To aid in the annotation of the producer process, we define the invariant

$$I = \forall m \in \sigma_F : \mathcal{I}(m).$$

The predicate, $\mathcal{I}(m)$, describes the state of the implicit variables at the transmission of message m. As shown in Figure 3.2, m is the composite

$$m = <m.type, m.arnum, m.index, m.value>,$$

where *m.type* is the type of the message, *m.arnum* is the number of the array (1 or 2), *m.index* is the index in the array, and *m.value* is the element of the array at *m.index*. The annotation of the producer must establish the structural properties of the receipt order—the satisfaction and non-interference proofs explicitly require this information. On a message-by-message basis, I states those required structural properties. Given that there is only a single F-channel in this example, we drop the F subscript on the relevant implicit variables.

The form of I deserves some discussion since it is the entity by which allowable message receipt order is factored into our reasoning. As each message is transmitted, it is incorporated in the implicit variables to describe the subsequent communication state of the F-channel. The transmission of a new message, m, however, will not remove any previously transmitted message from σ; nor will it remove any previously established edge (message pair) from $\prec$. The only possible change in σ is the addition of the newly transmitted message. The only possible changes in the structure of $\prec$ are new links between elements of the set representing the backward flush point or of the set representing free messages (as this set was just before the transmission of m) and m itself. Anything which was asserted about σ and $\prec$ before the transmission of m, must still be true after the transmission of m. If m' is the message transmitted immediately before m and if $\mathcal{I}(m')$ is true, then $\mathcal{I}(m')$ will be true after the transmission of m. Now, however, $\mathcal{I}(m)$ will be true also. It is easy to extend this argument inductively to see that I follows.

The form of I clearly restricts the form of $\mathcal{I}$. It must be parameterized in such a way that it remains true even after subsequent messages are transmitted. Absolute statements, say $|\sigma(m)| = 6$, would not be valid. In the examples which follow, we take advantage of our knowledge of the receipt-order relation to state the values of the implicit variables just after the transmission of message m parametrically in terms of m itself. As will be shown explicitly in the detailed proof of the producer/consumer system below, I is initially vacuously true in the producer; as each message is transmitted by the producer, I is shown to be preserved. In the consumer, we show that no local action or communication invalidates I, and thus, I is treated as a *global* invariant of the system.

We explain our use of I in contrast with Schlichting and Schneider's treatment of virtual circuits [SS84]. The receipt order for a virtual circuit is much simpler than that for an F-channel—it is the same as the transmission order. The virtual circuit receipt-order restriction is *static* in the sense that the sender cannot specify alternative receipt orders. There is a single receipt order; the messages in the send multiset are totally ordered by time of transmission. This fact is explicitly used in their proofs through the operators on sequences. In a sense, their methodology implicitly uses an I which states that (among other things) if message m is transmitted before message m', then m will be received before m'.

An F-channel allows as many receipt orders as there are distinct topological sorts of $\prec$. Further, the sending process "builds" $\prec$ as it sends messages. In this sense, the receipt order for an F-channel is *dynamic*: it is not known before the sender executes, and it is constructed incrementally as the sender transmits successive messages. I is sufficiently weak to capture the complexity of the F-channel in a predicate which is globally true. We rely upon I (and the F-channel network axiom) in the receiving process to manage explicitly the complex receipt-order requirements.

As mentioned, $\mathcal{I}(m)$ describes the state of the implicit variables at the transmission of m. We let the state of the implicit variables just following the send of message m be denoted as $\sigma(m)$, $\prec(m)$, $\tau(m)$, and $\Omega(m)$.

$$\mathcal{I}(m) \equiv \mathcal{I}_{\sigma(m)} \wedge \mathcal{I}_{\prec(m)} \wedge \mathcal{I}_{\tau(m)} \wedge \mathcal{I}_{\Omega(m)},$$

where

$$
\begin{aligned}
\mathcal{I}_{\sigma(m)} \equiv\ & (<m.type, m.arnum> = <\mathbf{2F}, 1> \Rightarrow (\sigma(m) = D,\\
& \quad \text{where } D = \{<\mathbf{2F}, 1, 0, 0>\}))\\
& \wedge (<m.type, m.arnum> = <\mathbf{BF}, 1> \Rightarrow (\sigma(m) = D \oplus E,\\
& \quad \text{where } E = \{<\mathbf{BF}, 1, a, 0>\}))\\
& \wedge (<m.type, m.arnum> = <\mathbf{Ord}, 1> \Rightarrow (\sigma(m) = D \oplus E \oplus F(m.index),\\
& \quad \text{where} F(x) = \{\mu \mid \forall i : 0 < i \le x :: <\mathbf{Ord}, 1, i, A[i]>\}))\\
& \wedge (<m.type, m.arnum> = <\mathbf{FF}, 1> \Rightarrow (\sigma(m) = D \oplus E \oplus F(a) \oplus G,\\
& \quad \text{where } G = \{<\mathbf{FF}, 1, a, 0>\}))\\
& \wedge (<m.type, m.arnum> = <\mathbf{BF}, 2> \Rightarrow (\sigma(m) = D \oplus E \oplus F(a) \oplus G \oplus H,\\
& \quad \text{where} H = \{<\mathbf{BF}, 2, b, 0>\}))\\
& \wedge (<m.type, m.arnum> = <\mathbf{Ord}, 2> \Rightarrow (\sigma(m) = D \oplus E \oplus F(a) \oplus G \oplus H \oplus J(m.index),\\
& \quad \text{where} J(x) = \{\mu \mid \forall i : 0 < i \le x :: <\mathbf{Ord}, 2, i, B[i]>\}))\\
& \wedge (<m.type, m.arnum> = <\mathbf{FF}, 2> \Rightarrow (\sigma(m) = D \oplus E \oplus F(a) \oplus G \oplus H \oplus J(b) \oplus K,\\
& \quad \text{where} K = \{<\mathbf{FF}, 2, b, 0>\}))\\
& \wedge (<m.type, m.arnum> = <\mathbf{2F}, 2> \Rightarrow (\sigma(m) = D \oplus E \oplus F(a) \oplus G \oplus H \oplus J(b) \oplus K \oplus L,\\
& \quad \text{where} L = \{<\mathbf{2F}, 2, 0, 0>\}));
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{I}_{\prec(m)} \equiv\ & (<m.type, m.arnum> = <\mathbf{BF}, 1> \Rightarrow (\prec(m) = N,\\
& \quad \text{where } N = \{(\mu_1, \mu_2) \mid \mu_1 = <\mathbf{2F}, 1, 0, 0> \wedge \mu_2 = <\mathbf{BF}, 1, a, 0>\}))\\
& \wedge (<m.type, m.arnum> = <\mathbf{Ord}, 1> \Rightarrow (\prec(m) = N \oplus O(m.index),\\
& \quad \text{where } O(x) = \{(\mu_1, \mu_2) \mid \forall i : 0 < i \le x :: \mu_1 = <\mathbf{BF}, 1, a, 0>\\
& \qquad \wedge \mu_2 = <\mathbf{Ord}, 1, i, A[i]>\}))\\
& \wedge (<m.type, m.arnum> = <\mathbf{FF}, 1> \Rightarrow (\prec(m) = N \oplus O(a) \oplus P,\\
& \quad \text{where } P = \{(\mu_1, \mu_2) \mid \forall i : 0 < i \le a :: \mu_1 = <\mathbf{Ord}, 1, i, A[i]>\\
& \qquad \wedge \mu_2 = <\mathbf{FF}, 1, a, 0>\}))\\
& \wedge (<m.type, m.arnum> = <\mathbf{BF}, 2> \Rightarrow (\prec(m) = N \oplus O(a) \oplus P \oplus Q,\\
& \quad \text{where } Q = \{(\mu_1, \mu_2) \mid \mu_1 = <\mathbf{BF}, 1, a, 0> \wedge \mu_2 = <\mathbf{BF}, 2, b, 0>\}))\\
& \wedge (<m.type, m.arnum> = <\mathbf{Ord}, 2> \Rightarrow (\prec(m) = N \oplus O(a) \oplus P \oplus Q \oplus R(m.index),\\
& \quad \text{where } R(x) = \{(\mu_1, \mu_2) \mid \forall i : 0 < i \le x :: \mu_1 = <\mathbf{BF}, 2, b, 0> \wedge \mu_2 = <\mathbf{Ord}, 2, i, B[i]>\}))\\
& \wedge (<m.type, m.arnum> = <\mathbf{FF}, 2> \Rightarrow (\prec(m) = N \oplus O(a) \oplus P \oplus Q \oplus R(b) \oplus S,\\
& \quad \text{where } S = \{(\mu_1, \mu_2) \mid (\forall i : 0 < i \le b :: \mu_1 = <\mathbf{Ord}, 2, i, B[i]> \wedge \mu_2 = <\mathbf{FF}, 2, b, 0>)\\
& \qquad \vee (\mu_1 = <\mathbf{FF}, 1, a, 0> \wedge \mu_2 = <\mathbf{FF}, 2, b, 0>)\}))\\
& \wedge (<m.type, m.arnum> = <\mathbf{2F}, 2> \Rightarrow (\prec(m) = N \oplus O(a) \oplus P \oplus Q \oplus R(b) \oplus S \oplus T,\\
& \quad \text{where } T = \{(\mu_1, \mu_2) \mid \mu_1 = <\mathbf{FF}, 2, b, 0> \wedge \mu_2 = <\mathbf{2F}, 2, 0, 0>\}));
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{I}_{\tau(m)} \equiv\ & (<m.type, m.arnum> = <\mathbf{2F}, 1> \Rightarrow \tau(m) = \{<\mathbf{2F}, 1, 0, 0>\}) \wedge\\
& ((m.type \ne \mathbf{2F} \wedge m.arnum = 1) \Rightarrow \tau(m) = \{<\mathbf{BF}, 1, a, 0>\}) \wedge\\
& ((m.type \ne \mathbf{2F} \wedge m.arnum = 2) \Rightarrow \tau(m) = \{<\mathbf{BF}, 2, b, 0>\}) \wedge\\
& (<m.type, m.arnum> = <\mathbf{2F}, 2> \Rightarrow \tau(m) = \{<\mathbf{2F}, 2, 0, 0>\});
\end{aligned}
$$

$$\begin{aligned}
\mathcal{I}_{\Omega(m)} \equiv\ & ((m.type \neq \mathbf{Ord} \wedge m.arnum = 1) \Rightarrow \Omega(m) \\
& \quad = \{<m.type, m.arnum, m.index, m.data>\}) \\
& \wedge\ (<m.type, m.arnum> = <\mathbf{Ord}, 1> \Rightarrow \Omega(m) = \{\mu \mid \forall i : 0 < i \leq m.index :: \\
& \quad <\mathbf{Ord}, 1, i, A[i]>\}) \\
& \wedge\ (<m.type, m.arnum> = <\mathbf{BF}, 2> \Rightarrow \Omega(m) = \{<\mathbf{FF}, 1, a, 0>, <\mathbf{BF}, 2, b, 0>\}) \\
& \wedge\ (<m.type, m.arnum> = <\mathbf{Ord}, 2> \Rightarrow \Omega(m) = \{<\mathbf{FF}, 1, a, 0> \\
& \quad \wedge\ (\mu \mid \forall i : 0 < i \leq m.index :: <\mathbf{Ord}, 2, i, B[i]>)\}) \\
& \wedge\ ((<m.type, m.arnum> = <\mathbf{FF}, 2>\ \vee\ <m.type, m.arnum> = <\mathbf{2F}, 2>) \\
& \quad \Rightarrow \Omega(m) = \{<m.type, m.arnum, m.index, m.data>\});
\end{aligned}$$

This relatively intimidating set of assertions merely states, in tedious but complete logical terms, the state of the implicit variables just after the transmission of message m by the producer process. For example, the first conjunct of $\mathcal{I}_{\prec(m)}$ describes the edge in $\prec$ as it appears immediately after the transmission of the first backward flush message. The second conjunct defines those edges which go from this backward flush message to the ordinary messages that transmit the first array. The third conjunct defines those edges which go from these ordinary messages to the forward flush message; a message that signals the end of the array transmission. It is merely a restatement of what is drawn in Figure 3.2.

```
PROD:: var A      : array 1..M of integer;
           B      : array 1..N of integer;
           a,b    : integer;
           i      : integer;

                  ... PREVIOUS APPLICATIONS ...
                  { $I$ }
       s1: send (2F, 1, 0, 0) on F;
                  { $I$ }
       find(a);
                  { $a \leq M \wedge I$ }
       s2: send (BF, 1, a, 0) on F;
                  { $a \leq M \wedge I$ }
       i := 0;
                  { $i = 0 \wedge a \leq M \wedge I$ }
       while i < a do
                  { $i < a \wedge a \leq M \wedge I$ }
          i := i + 1;
                  { $i \leq a \wedge a \leq M \wedge I$ }
          s3: send (Ord, 1, i, A[i]) on F;
                  { $i \leq a \wedge a \leq M \wedge I$ }
```

```
od;
            $\{ i = a \wedge a \leq M \wedge I \}$
s4: send (FF, 1, a, 0) on F;
            $\{ i = a \wedge a \leq M \wedge I \}$
find(b);
            $\{ b \leq N \wedge i = a \wedge$
s5: send (BF, 2, b, 0) on F;
            $\{ b \leq N \wedge i = a$
i := 0;
            $\{ i = 0 \wedge b \leq$
while i < b do
            $\{ i < b \wedge b \leq$
    i := i + 1;
            $\{ i \leq b \wedge b \leq$
    s6: send (Ord, 2, i, B[i]) on F;
            $\{ i \leq b \wedge b \leq$
od;
            $\{ i = b \wedge b \leq$
s7: send (FF, 2, b, 0) on F;
            $\{ i = b \wedge b \leq$
s8: send (2F, 2, 0, 0) on F;
            $\{ i = b \wedge b \leq$
            ... FOLLOWING APPLICATIONS ...
```

The proof in isolation of the producer is straightforward. The invariant truth of I is established as part of that proof. Initially, it is trivially true. As successive messages are sent, I is inductively validated through application of the Send Axiom apropos of the type of message being transmitted. As an example, consider the **send** statement labeled s4 in the code of the producer. We need to prove

$$\{\mathrm{i} = \mathrm{a} \wedge \mathrm{a} \leq \mathrm{M} \wedge I\} \ \mathbf{s4} \ \{\mathrm{i} = \mathrm{a} \wedge \mathrm{a} \leq \mathrm{M} \wedge I\}.$$

Given that $\mathrm{i} = \mathrm{a} \wedge \mathrm{a} \leq \mathrm{M}$ follows directly from the precondition, we concentrate on I and the semantics of the transmission of the forward flush message according to the Forward Flush Send Axiom. The semantic effect of s4 is that message $m = <\mathbf{FF}, 1, a, 0>$ is transmitted on F. More precisely, we must show that

$$\{I^{\sigma(m), \quad \prec(m), \quad \Omega(m)}_{\sigma(m)\oplus\{m\}, \prec(m)\oplus C(m), \{m\}}\} \quad \mathbf{s4} \quad \{I\}.$$

We can show this easily. We may check every conjunct of $\mathcal{I}(m)$ for every message in $\sigma(m)$ to validate that I is, in fact, preserved across s4.

In the consumer, the invariant truth of I follows directly from the truth of I as established by the producer and the fact that no action in the consumer affects any variable used in I. We defer the non-interference aspects of this claim until later in this section.

Defining the set of messages which have been consumed thus far (in terms of the consumer's variables X, C, and D) is helpful:

$$\begin{aligned}
\mathcal{C}(\mathrm{X},\mathrm{C},\mathrm{D}) \equiv \{\mu \mid\ & (\mu = <\mathbf{2F}, i, 0, 0> \wedge i \in \mathrm{X}[1]) \ \vee \ (\mu = <\mathbf{BF}, i, j, 0> \wedge i \in \mathrm{X}[2]) \\
& \vee(\mu = <\mathbf{Ord}, 1, i, \mathrm{C}[i]> \wedge i \in \mathrm{X}[3]) \vee (\mu = <\mathbf{Ord}, 2, i, \mathrm{D}[i]> \wedge i \in \mathrm{X}[4]) \\
& \vee(\mu = <\mathbf{FF}, i, j, 0> \wedge i \in \mathrm{X}[5])\}.
\end{aligned}$$

A newly received message (with its type assigned to the consumer's variable mtype, its array number assigned to marnum, its index assigned to mindex, and its value assigned to mvalue) must satisfy the following parametric assertion:

$$\begin{aligned}
&\mathcal{N}(\mathrm{X},\mathrm{C},\mathrm{D}) \equiv (\mathrm{mtype} = \mathbf{2F} \wedge \mathrm{marnum} \notin \mathrm{X}[1] \wedge ((\mathrm{marnum} = 1 \wedge \forall i : 1 \leq i \leq 5 :: \mathrm{X}[i] = \phi) \\
&\quad \vee(\mathrm{marnum} = 2 \wedge |\mathrm{X}[1]| = 1 \wedge |\mathrm{X}[2]| = 2 \wedge |\mathrm{X}[3]| = \mathrm{c} \wedge |\mathrm{X}[4]| = \mathrm{d} \wedge |\mathrm{X}[5]| = 2))) \\
&\vee(\mathrm{mtype} = \mathbf{BF} \wedge \mathrm{marnum} \notin \mathrm{X}[2] \wedge ((\mathrm{marnum} = 1 \wedge \mathrm{mindex} \leq \mathrm{M} \wedge |\mathrm{X}[1]| = 1 \\
&\quad \wedge \forall i : 2 \leq i \leq 5 :: \mathrm{X}[i] = \phi) \\
&\vee(\mathrm{marnum} = 2 \wedge \mathrm{mindex} \leq \mathrm{N} \wedge |\mathrm{X}[1]| = 1 \wedge |\mathrm{X}[2]| = 1 \wedge |\mathrm{X}[3]| \leq \mathrm{c} \wedge \mathrm{X}[4] = \phi \wedge |\mathrm{X}[5]| \leq 1))) \\
&\vee(\mathrm{mtype} = \mathbf{Ord} \\
&\quad \wedge((\mathrm{marnum} = 1 \wedge \mathrm{mindex} \notin \mathrm{X}[3] \wedge |\mathrm{X}[1]| = 1 \wedge 1 \leq |\mathrm{X}[2]| \leq 2 \wedge |\mathrm{X}[3]| < \mathrm{c} \\
&\qquad\qquad \wedge|\mathrm{X}[4]| \leq \mathrm{d} \wedge \mathrm{X}[5] = \phi) \\
&\quad \vee(\mathrm{marnum} = 2 \wedge \mathrm{mindex} \notin \mathrm{X}[4] \wedge |\mathrm{X}[1]| = 1 \wedge |\mathrm{X}[2]| = 2 \wedge |\mathrm{X}[3]| \leq \mathrm{c} \\
&\qquad\qquad \wedge|\mathrm{X}[4]| < \mathrm{d} \wedge |\mathrm{X}[5]| \leq 1))) \\
&\vee(\mathrm{mtype} = \mathbf{FF} \wedge \mathrm{marnum} \notin \mathrm{X}[5] \\
&\quad \wedge((\mathrm{marnum} = 1 \wedge |\mathrm{X}[1]| = 1 \wedge 1 \leq |\mathrm{X}[2]| \leq 2 \wedge |\mathrm{X}[3]| = \mathrm{c} \wedge |\mathrm{X}[4]| \leq \mathrm{d} \wedge \mathrm{X}[5] = \phi) \\
&\quad \vee(\mathrm{marnum} = 2 \wedge |\mathrm{X}[1]| = 1 \wedge |\mathrm{X}[2]| = 2 \wedge |\mathrm{X}[3]| = \mathrm{c} \wedge |\mathrm{X}[4]| = \mathrm{d} \wedge |\mathrm{X}[5]| = 1))).
\end{aligned}$$

The above assertion follows from our understanding of the structure of the receipt-order relation. It is miraculous in the proof in isolation of the consumer. Establishing its truth is the primary task in the satisfaction proof.

```
CONS:: var C                : array 1..M of integer;
           D                : array 1..N of integer;
           c,d              : integer;
           mtype            : { 2F, BF, Ord, FF};
           marnum           : integer;
           mindex, mvalue   : integer;
           j                : integer;
           done             : boolean;
           sumC, sumD       : integer;

                            ... PREVIOUS APPLICATIONS ...
$\{ \rho = \emptyset \wedge I \}$
        C, D, sumC, sumD, done, X := $\emptyset$, $\emptyset$, 0, 0, $false$, $\Phi$;
$\{ C = \emptyset \wedge D = \emptyset \wedge sumC = 0 \wedge sumD = 0 \wedge done = false \wedge X = \Phi \wedge \rho = \emptyset \wedge I \}$
        while not done do
$\{ done = false \wedge \rho = \mathcal{C}(X, C, D) \wedge I \}$
            r1: receive (mtype, marnum, mindex, mvalue) from F;
$\{ \mathcal{N}(X, C, D) \wedge done = false \wedge \rho = \mathcal{C}(X, C, D) \oplus \{<mtype, marnum, mindex, mvalue>\} \wedge I \}$
            case mtype of
                2F:  X[1] := X[1] $\cup$ {marnum};
$\{ done = false \wedge mtype = 2F \wedge marnum \in X[1] \wedge ((marnum = 1 \wedge |X[1]| = 1 \wedge \forall i : 2 \leq i \leq 5 :: X[i] = \phi)$
    $\vee (marnum = 2 \wedge |X[1]| = 2 \wedge |X[2]| = 2 \wedge |X[3]| = c \wedge |X[4]| = d \wedge |X[5]| = 2)) \wedge \rho = \mathcal{C}(X, C, D) \wedge I \}$
                    if marnum = 2 then
$\{ done = false \wedge mtype = 2F \wedge marnum \in X[1] \wedge marnum = 2 \wedge |X[1]| = 2 \wedge |X[2]| = 2$
    $\wedge |X[3]| = c \wedge |X[4]| = d \wedge |X[5]| = 2 \wedge \rho = \mathcal{C}(X, C, D) \wedge I \}$
                        done := true;
$\{ done = true \wedge mtype = 2F \wedge marnum \in X[1] \wedge marnum = 2 \wedge |X[1]| = 2 \wedge |X[2]| = 2$
    $\wedge |X[3]| = c \wedge |X[4]| = d \wedge |X[5]| = 2 \wedge \rho = \mathcal{C}(X, C, D) \wedge I \}$
                    fi;
                BF:  X[2] := X[2] $\cup$ {marnum};
$\{ done = false \wedge mtype = BF \wedge marnum \in X[2] \wedge \rho = \mathcal{C}(X, C, D) I$
    $\wedge ((marnum = 1 \wedge mindex \leq M \wedge |X[1]| = 1 \wedge |X[2]| = 1 \wedge \forall i : 3 \leq i \leq 5 :: X[i] = \phi)$
    $\vee (marnum = 2 \wedge mindex \leq N \wedge |X[1]| = 1 \wedge |X[2]| = 2 \wedge |X[3]| \leq c \wedge X[4] = \phi \wedge |X[5]| \leq 1)) \}$
                    if marnum = 1 then
$\{ done = false \wedge mtype = BF \wedge marnum \in X[2] \wedge marnum = 1 \wedge mindex \leq M$
    $\wedge |X[1]| = 1 \wedge |X[2]| = 1 \wedge \forall i : 3 \leq i \leq 5 :: X[i] = \phi \wedge \rho = \mathcal{C}(X, C, D) \wedge I \}$
                        c := mindex
```

```
{ $c = \text{mindex} \wedge \text{done} = \mathit{false} \wedge \text{mtype} = \textbf{BF} \wedge \text{marnum} \in X[2] \wedge \text{marnum} = 1 \wedge \text{mindex} \leq M$
      $\wedge|X[1]| = 1 \wedge |X[2]| = 1 \wedge \forall i : 3 \leq i \leq 5 :: X[i] = \phi \wedge \rho = \mathcal{C}(X, C, D) \wedge I$ }
                    else
{ $\text{done} = \mathit{false} \wedge \text{mtype} = \textbf{BF} \wedge \text{marnum} \in X[2] \wedge \text{marnum} = 2 \wedge \text{mindex} \leq N$
      $\wedge|X[1]| = 1 \wedge |X[2]| = 2 \wedge |X[3]| \leq c \wedge X[4] = \phi \wedge |X[5]| \leq 1 \wedge \rho = \mathcal{C}(X, C, D) \wedge I$ }
                    d := mindex;
{ $d = \text{mindex} \wedge \text{done} = \mathit{false} \wedge \text{mtype} = \textbf{BF} \wedge \text{marnum} \in X[2] \wedge \text{marnum} = 2 \wedge \text{mindex} \leq N$
      $\wedge|X[1]| = 1 \wedge |X[2]| = 2 \wedge |X[3]| \leq c \wedge X[4] = \phi \wedge |X[5]| \leq 1 \wedge \rho = \mathcal{C}(X, C, D) \wedge I$ }
                    fi;
                Ord: if marnum = 1 then
{ $\text{done} = \mathit{false} \wedge \text{mtype} = \textbf{Ord} \wedge \text{marnum} = 1 \wedge \text{mindex} \notin X[3] \wedge |X[1]| = 1 \wedge 1 \leq |X[2]| \leq 2$
      $\wedge|X[3]| < c \wedge |X[4]| \leq d \wedge X[5] = \phi \wedge \rho = \mathcal{C}(X, C, D) \oplus \{<\text{mtype}, \text{marnum}, \text{mindex}, \text{mvalue}>\} \wedge I$ }
                    C[mindex], X[3] := mvalue, X[3] ∪ {mindex}
{ $C[\text{mindex}] = \text{mvalue} \wedge \text{done} = \mathit{false} \wedge \text{mtype} = \textbf{Ord} \wedge \text{marnum} = 1 \wedge \text{mindex} \in X[3]$
      $\wedge|X[1]| = 1 \wedge 1 \leq |X[2]| \leq 2 \wedge |X[3]| \leq c \wedge |X[4]| \leq d \wedge X[5] = \phi \wedge \rho = \mathcal{C}(X, C, D) \wedge I$ }
                    else
{ $\text{done} = \mathit{false} \wedge \text{mtype} = \textbf{Ord} \wedge \text{marnum} = 2 \wedge \text{mindex} \notin X[4] \wedge |X[1]| = 1 \wedge |X[2]| = 2$
      $\wedge|X[3]| \leq c \wedge |X[4]| < d \wedge |X[5]| \leq 1 \wedge \rho = \mathcal{C}(X, C, D) \oplus \{<\text{mtype}, \text{marnum}, \text{mindex}, \text{mvalue}>\} \wedge I$ }
                    D[mindex], X[4] := mvalue, X[4] ∪ {mindex};
{ $D[\text{mindex}] = \text{mvalue} \wedge \text{done} = \mathit{false} \wedge \text{mtype} = \textbf{Ord} \wedge \text{marnum} = 2 \wedge \text{mindex} \in X[4]$
      $\wedge|X[1]| = 1 \wedge |X[2]| = 2 \wedge |X[3]| \leq c \wedge |X[4]| \leq d \wedge |X[5]| \leq 1 \wedge \rho = \mathcal{C}(X, C, D) \wedge I$ }
                    fi;
                FF: j, X[5] := 0, X[5] ∪ {marnum};
{ $j = 0 \wedge \text{done} = \mathit{false} \wedge \text{mtype} = \textbf{FF} \wedge \text{marnum} \in X[5] \wedge \rho = \mathcal{C}(X, C, D) \wedge I$
      $\wedge((\text{marnum} = 1 \wedge |X[1]| = 1 \wedge 1 \leq |X[2]| \leq 2 \wedge |X[3]| = c \wedge |X[4]| \leq d \wedge |X[5]| = 1)$
      $\vee(\text{marnum} = 2 \wedge |X[1]| = 1 \wedge |X[2]| = 2 \wedge |X[3]| = c \wedge |X[4]| = d \wedge |X[5]| = 2))$ }
                    if marnum = 1 then
{ $j = 0 \wedge \text{done} = \mathit{false} \wedge \text{mtype} = \textbf{FF} \wedge \text{marnum} \in X[5] \wedge \text{marnum} = 1 \wedge |X[1]| = 1 \wedge 1 \leq |X[2]| \leq 2$
      $\wedge|X[3]| = c \wedge |X[4]| \leq d \wedge |X[5]| = 1 \wedge \rho = \mathcal{C}(X, C, D) \wedge I$ }
                        while j < mindex do
{ $j < \text{mindex} \wedge \text{sumC} = \sum_{i=1}^{j} C[i] \wedge \text{done} = \mathit{false} \wedge \text{mtype} = \textbf{FF} \wedge \text{marnum} \in X[5] \wedge \text{marnum} = 1$
      $\wedge|X[1]| = 1 \wedge 1 \leq |X[2]| \leq 2 \wedge |X[3]| = c \wedge |X[4]| \leq d \wedge |X[5]| = 1 \wedge \rho = \mathcal{C}(X, C, D) \wedge I$ }
                            j := j + 1;
{ $j \leq \text{mindex} \wedge \text{sumC} = \sum_{i=1}^{j-1} C[i] \wedge \text{done} = \mathit{false} \wedge \text{mtype} = \textbf{FF} \wedge \text{marnum} \in X[5] \wedge \text{marnum} = 1$
      $\wedge|X[1]| = 1 \wedge 1 \leq |X[2]| \leq 2 \wedge |X[3]| = c \wedge |X[4]| \leq d \wedge |X[5]| = 1 \wedge \rho = \mathcal{C}(X, C, D) \wedge I$ }
                            sumC := sumC + C[j];
{ $j \leq \text{mindex} \wedge \text{sumC} = \sum_{i=1}^{j} C[i] \wedge \text{done} = \mathit{false} \wedge \text{mtype} = \textbf{FF} \wedge \text{marnum} \in X[5] \wedge \text{marnum} = 1$
      $\wedge|X[1]| = 1 \wedge 1 \leq |X[2]| \leq 2 \wedge |X[3]| = c \wedge |X[4]| \leq d \wedge |X[5]| = 1 \wedge \rho = \mathcal{C}(X, C, D) \wedge I$ }
                        od
                    else
{ $j = 0 \wedge \text{done} = \mathit{false} \wedge \text{mtype} = \textbf{FF} \wedge \text{marnum} \in X[5] \wedge \text{marnum} = 2$
      $\wedge|X[1]| = 1 \wedge |X[2]| = 2 \wedge |X[3]| = c \wedge |X[4]| = d \wedge |X[5]| = 2 \wedge \rho = \mathcal{C}(X, C, D) \wedge I$ }
                        while j < mindex do
{ $j < \text{mindex} \wedge \text{sumD} = \sum_{i=1}^{j} D[i] \wedge \text{done} = \mathit{false} \wedge \text{mtype} = \textbf{FF} \wedge \text{marnum} \in X[5] \wedge \text{marnum} = 2$
      $\wedge|X[1]| = 1 \wedge |X[2]| = 2 \wedge |X[3]| = c \wedge |X[4]| = d \wedge |X[5]| = 2 \wedge \rho = \mathcal{C}(X, C, D) \wedge I$ }
```

```
                                        j := j + 1;
{ $j \leq$ mindex $\wedge$ sumD $= \sum_{i=1}^{j-1}$ D[i] $\wedge$ done $= false \wedge$ mtype $=$ FF $\wedge$ marnum $\in$ X[5] $\wedge$ marnum $= 2$
      $\wedge|$X[1]$| = 1 \wedge |$X[2]$| = 2 \wedge |$X[3]$| =$ c $\wedge |$X[4]$| =$ d $\wedge |$X[5]$| = 2 \wedge \rho = \mathcal{C}$(X, C, D) $\wedge I$ }
                                        sumD := sumD + D[j];
{ $j \leq$ mindex $\wedge$ sumD $= \sum_{i=1}^{j}$ D[i] $\wedge$ done $= false \wedge$ mtype $=$ FF $\wedge$ marnum $\in$ X[5] $\wedge$ marnum $= 2$
      $\wedge|$X[1]$| = 1 \wedge |$X[2]$| = 2 \wedge |$X[3]$| =$ c $\wedge |$X[4]$| =$ d $\wedge |$X[5]$| = 2 \wedge \rho = \mathcal{C}$(X, C, D) $\wedge I$ }
                                   od;
                              fi;
        od;
{ done $= true \wedge |$X[1]$| = 2 \wedge |$X[2]$| = 2 \wedge |$X[3]$| =$ c $\wedge |$X[4]$| =$ d $\wedge |$X[5]$| = 2 \wedge \rho = \mathcal{C}$(X, C, D) $\wedge I$}
                         ... FOLLOWING APPLICATIONS ...
```

We omit the proof in isolation of the consumer as it is straightforward. The invariant truth of I is detailed in the non-interference discussion later in this section. We justify the miraculous postcondition of the **receive** next. Let $MTEXT$ denote the message which is being received, i.e., <mtype, marnum, mindex, mvalue>.

The antecedent in the satisfaction proof is

$$\text{pre}(\mathbf{r1}) \wedge (MTEXT \in \sigma \ominus \rho) \wedge (\forall m : m \in \sigma \wedge m \not\prec MTEXT :: m \in \rho).$$

We may substitute for pre(**r1**), yielding

$$(\text{done} = false \wedge \rho = \mathcal{C}(\text{X}, \text{C}, \text{D}) \wedge I) \wedge (MTEXT \in \sigma \ominus \rho) \wedge (\forall m : m \in \sigma \wedge m \not\prec MTEXT :: m \in \rho).$$

Our obligation in the satisfaction proof is to establish the truth of the following consequent assuming the previously stated antecedent:

$$\text{post}(\mathbf{r1})^{<\text{mtype,marnum,mindex,mvalue}>,\ \rho}_{MTEXT,\qquad\qquad \rho \oplus \{MTEXT\}},$$

or equivalently, we need to show

$$(\mathcal{N}(\text{X}, \text{C}, \text{D}) \ \wedge \ \text{done} = false \ \wedge \ \rho = \mathcal{C}(\text{X}, \text{C}, \text{D})$$
$$\oplus\{<\text{mtype}, \text{marnum}, \text{mindex}, \text{mvalue}>\} \wedge I)^{<\text{mtype,marnum,mindex,mvalue}>,\ \rho}_{MTEXT,\qquad\qquad \rho \oplus \{MTEXT\}}.$$

Given that (done $= false \wedge \rho = \mathcal{C}(\mathrm{X, C, D}) \wedge I$) is in the antecedent, then clearly

$$(\text{done} = false \wedge \rho \oplus \{MTEXT\} = \mathcal{C}(\mathrm{X,C,D}) \oplus \{MTEXT\} \wedge I).$$

Therefore, we concentrate on $\mathcal{N}(\mathrm{X,C,D})$.

When the **receive** statement executes, we may ascertain several facts about the transmission of $MTEXT$ (i.e., the message being added to ρ). The field mtype can be any of the four message types. This follows from $\mathcal{I}_{\sigma(MTEXT)}$ which we know must be true for every element of ρ from the truth of I and the fact that the en route property requires $\rho \subseteq \sigma$. Suppose that mtype is **2F**. From the en route property and $\mathcal{I}_{\sigma(MTEXT)}$, we know that <mtype, marnum, mindex, mvalue> has not been received before and, therefore, marnum is not an element of X[1]. We also know, based on the order property and $\mathcal{I}_{\prec(MTEXT)}$, that this two-way flush is either the first or the last message to be received. This fact allows us to specify the number of values that must be in each element of X. In sum,

$$\begin{aligned}(MTEXT = {<}\mathbf{2F}, p, 0, 0{>} \wedge p \notin \mathrm{X}[1] \wedge ((p = 1 \wedge \forall i : 1 \leq i \leq 5 :: \mathrm{X}[i] = \phi)\\ \vee(p = 2 \wedge |\mathrm{X}[1]| = 1 \wedge |\mathrm{X}[2]| = 2 \wedge |\mathrm{X}[3]| = \mathrm{c} \wedge |\mathrm{X}[4]| = \mathrm{d} \wedge |\mathrm{X}[5]| = 2))).\end{aligned}$$

If we assume the message being received is a backward flush, then we can show facts about this message as well. First, as in the previous case, marnum is not an element of X[2]; this follows from the fact the message has not been received before. Second, if the value of marnum is one, then, based on the order property and $\mathcal{I}_{\prec(MTEXT)}$, this is the second message to be received (following the previous two-way flush message). If, on the other hand, the value of marnum is two, then based on the order property and $\mathcal{I}_{\prec(MTEXT)}$, this message is received after the first backward flush; ordinary messages and the forward flush for the first array may be received before or after this second backward flush. These facts lead us to the following clause:

$(MTEXT = <\mathbf{BF}, q, v, 0> \wedge q \notin \mathrm{X}[2] \wedge ((q = 1 \wedge v \leq \mathrm{M} \wedge |\mathrm{X}[1]| = 1 \wedge \forall i : 2 \leq i \leq 5 :: \mathrm{X}[i] = \phi)$
$\quad \vee (q = 2 \wedge v \leq \mathrm{N} \wedge |\mathrm{X}[1]| = 1 \wedge |\mathrm{X}[2]| = 1 \wedge |\mathrm{X}[3]| \leq \mathrm{c} \wedge \mathrm{X}[4] = \phi \wedge |\mathrm{X}[5]| \leq 1))).$

The index from the receipt of an ordinary message, based on the en route property and $\mathcal{I}_{\sigma(MTEXT)}$, will not be an element of X[3] if the message pertains to the first array or will not be an element of X[4] if the message pertains to the second array. Suppose marnum is one. From the order property and $\mathcal{I}_{\prec(MTEXT)}$, we know the first two-way flush and the first backward flush must be received. We also know the second backward flush message may be received, but neither forward flush message can be received. If we suppose marnum is two, then we know, from the order property and $\mathcal{I}_{\prec(MTEXT)}$, the first two-way flush and both backward flush messages must be received, however, the first forward flush message may or may not be received. In other words,

$(MTEXT = <\mathbf{Ord}, s, t, u>$
$\quad \wedge((s = 1 \wedge t \notin \mathrm{X}[3] \wedge |\mathrm{X}[1]| = 1 \wedge 1 \leq |\mathrm{X}[2]| \leq 2 \wedge |\mathrm{X}[3]| < \mathrm{c} \wedge |\mathrm{X}[4]| \leq \mathrm{d} \wedge \mathrm{X}[5] = \phi)$
$\quad \vee(s = 2 \wedge t \notin \mathrm{X}[4] \wedge |\mathrm{X}[1]| = 1 \wedge |\mathrm{X}[2]| = 2 \wedge |\mathrm{X}[3]| \leq \mathrm{c} \wedge |\mathrm{X}[4]| < \mathrm{d} \wedge |\mathrm{X}[5]| \leq 1))).$

Lastly, the index from the receipt of a forward flush message, based again on the en route property and $\mathcal{I}_{\sigma(MTEXT)}$, cannot be an element of X[5]. Furthermore, based on the order property and $\mathcal{I}_{\prec(MTEXT)}$, the backward flush message and all the ordinary messages for the array ended by this forward flush must have been received.

$(MTEXT = <\mathbf{FF}, w, x, 0> \wedge w \notin \mathrm{X}[5]$
$\quad \wedge((w = 1 \wedge |\mathrm{X}[1]| = 1 \wedge 1 \leq |\mathrm{X}[2]| \leq 2 \wedge |\mathrm{X}[3]| = \mathrm{c} \wedge |\mathrm{X}[4]| \leq \mathrm{d} \wedge \mathrm{X}[5] = \phi)$
$\quad \vee(w = 2 \wedge |\mathrm{X}[1]| = 1 \wedge |\mathrm{X}[2]| = 2 \wedge |\mathrm{X}[3]| = \mathrm{c} \wedge |\mathrm{X}[4]| = \mathrm{d} \wedge |\mathrm{X}[5]| = 1))).$

Combining what we have established,

$$\begin{array}{l}
((MTEXT = \text{<}\mathbf{2F}, p, 0, 0\text{>} \wedge p \notin \mathrm{X}[1] \wedge ((p = 1 \wedge \forall i : 1 \leq i \leq 5 :: \mathrm{X}[i] = \phi) \\
\quad \vee (p = 2 \wedge |\mathrm{X}[1]| = 1 \wedge |\mathrm{X}[2]| = 2 \wedge |\mathrm{X}[3]| = \mathrm{c} \wedge |\mathrm{X}[4]| = \mathrm{d} \wedge |\mathrm{X}[5]| = 2))) \\
\vee (MTEXT = \text{<}\mathbf{BF}, q, v, 0\text{>} \wedge q \notin \mathrm{X}[2] \wedge ((q = 1 \wedge v \leq \mathrm{M} \wedge |\mathrm{X}[1]| = 1 \wedge \forall i : 2 \leq i \leq 5 :: \mathrm{X}[i] = \phi) \\
\quad \vee (q = 2 \wedge v \leq \mathrm{N} \wedge |\mathrm{X}[1]| = 1 \wedge |\mathrm{X}[2]| = 1 \wedge |\mathrm{X}[3]| \leq \mathrm{c} \wedge \mathrm{X}[4] = \phi \wedge |\mathrm{X}[5]| \leq 1))) \\
\vee (MTEXT = \text{<}\mathbf{Ord}, s, t, u\text{>} \\
\quad \wedge ((s = 1 \wedge t \notin \mathrm{X}[3] \wedge |\mathrm{X}[1]| = 1 \wedge 1 \leq |\mathrm{X}[2]| \leq 2 \wedge |\mathrm{X}[3]| < \mathrm{c} \wedge |\mathrm{X}[4]| \leq d \wedge \mathrm{X}[5] = \phi) \\
\quad \vee (s = 2 \wedge t \notin \mathrm{X}[4] \wedge |\mathrm{X}[1]| = 1 \wedge |\mathrm{X}[2]| = 2 \wedge |\mathrm{X}[3]| \leq \mathrm{c} \wedge |\mathrm{X}[4]| < d \wedge |\mathrm{X}[5]| \leq 1))) \\
\vee (MTEXT = \text{<}\mathbf{FF}, w, x, 0\text{>} \wedge w \notin \mathrm{X}[5] \\
\quad \wedge ((w = 1 \wedge |\mathrm{X}[1]| = 1 \wedge 1 \leq |\mathrm{X}[2]| \leq 2 \wedge |\mathrm{X}[3]| = \mathrm{c} \wedge |\mathrm{X}[4]| \leq d \wedge \mathrm{X}[5] = \phi) \\
\quad \vee (w = 2 \wedge |\mathrm{X}[1]| = 1 \wedge |\mathrm{X}[2]| = 2 \wedge |\mathrm{X}[3]| = \mathrm{c} \wedge |\mathrm{X}[4]| = d \wedge |\mathrm{X}[5]| = 1)))) \\
\wedge (\mathrm{done} = \mathit{false} \wedge \rho \oplus \{MTEXT\} = \mathcal{C}(\mathrm{X}, \mathrm{C}, \mathrm{D}) \oplus \{MTEXT\} \wedge I) \\
\equiv \\
((\text{<mtype, marnum, mindex, mvalue>} = \text{<}\mathbf{2F}, p, 0, 0\text{>} \wedge p \notin \mathrm{X}[1] \wedge ((p = 1 \wedge \forall i : 1 \leq i \leq 5 :: \mathrm{X}[i] = \phi) \\
\quad \vee (p = 2 \wedge |\mathrm{X}[1]| = 1 \wedge |\mathrm{X}[2]| = 2 \wedge |\mathrm{X}[3]| = \mathrm{c} \wedge |\mathrm{X}[4]| = \mathrm{d} \wedge |\mathrm{X}[5]| = 2))) \\
\vee (\text{<mtype, marnum, mindex, mvalue>} = \text{<}\mathbf{BF}, q, v, 0\text{>} \wedge q \notin \mathrm{X}[2] \\
\quad \wedge ((q = 1 \wedge v \leq \mathrm{M} \wedge |\mathrm{X}[1]| = 1 \wedge \forall i : 2 \leq i \leq 5 :: \mathrm{X}[i] = \phi) \\
\quad \vee (q = 2 \wedge v \leq \mathrm{N} \wedge |\mathrm{X}[1]| = 1 \wedge |\mathrm{X}[2]| = 1 \wedge |\mathrm{X}[3]| \leq \mathrm{c} \wedge \mathrm{X}[4] = \phi \wedge |\mathrm{X}[5]| \leq 1))) \\
\vee (\text{<mtype, marnum, mindex, mvalue>} = \text{<}\mathbf{Ord}, s, t, u\text{>} \\
\quad \wedge ((s = 1 \wedge t \notin \mathrm{X}[3] \wedge |\mathrm{X}[1]| = 1 \wedge 1 \leq |\mathrm{X}[2]| \leq 2 \wedge |\mathrm{X}[3]| < \mathrm{c} \wedge |\mathrm{X}[4]| \leq d \wedge \mathrm{X}[5] = \phi) \\
\quad \vee (s = 2 \wedge t \notin \mathrm{X}[4] \wedge |\mathrm{X}[1]| = 1 \wedge |\mathrm{X}[2]| = 2 \wedge |\mathrm{X}[3]| \leq \mathrm{c} \wedge |\mathrm{X}[4]| < d \wedge |\mathrm{X}[5]| \leq 1))) \\
\vee (\text{<mtype, marnum, mindex, mvalue>} = \text{<}\mathbf{FF}, w, x, 0\text{>} \wedge w \notin \mathrm{X}[5] \\
\quad \wedge ((w = 1 \wedge |\mathrm{X}[1]| = 1 \wedge 1 \leq |\mathrm{X}[2]| \leq 2 \wedge |\mathrm{X}[3]| = \mathrm{c} \wedge |\mathrm{X}[4]| \leq d \wedge \mathrm{X}[5] = \phi) \\
\quad \vee (w = 2 \wedge |\mathrm{X}[1]| = 1 \wedge |\mathrm{X}[2]| = 2 \wedge |\mathrm{X}[3]| = \mathrm{c} \wedge |\mathrm{X}[4]| = d \wedge |\mathrm{X}[5]| = 1)))) \\
\wedge (\mathrm{done} = \mathit{false} \wedge \rho = \mathcal{C}(\mathrm{X}, \mathrm{C}, \mathrm{D}) \\
\quad \oplus \{\text{<mtype, marnum, mindex, mvalue>}\} \wedge I)^{\text{<mtype,marnum,mindex,mvalue>},\ \rho}_{MTEXT,\qquad\qquad\qquad\qquad \rho \oplus \{MTEXT\}} . \\
\equiv \\
(\mathcal{N}(\mathrm{X}, \mathrm{C}, \mathrm{D}) \wedge \mathrm{done} = \mathit{false} \\
\quad \wedge\ \rho = \mathcal{C}(\mathrm{X}, \mathrm{C}, \mathrm{D}) \oplus \{\text{<mtype, marnum, mindex, mvalue>}\} \wedge I)^{\text{<mtype,marnum,mindex,mvalue>},\ \rho}_{MTEXT,\qquad\qquad\qquad\qquad \rho \oplus \{MTEXT\}} .
\end{array}$$

This is the required consequent, and hence, satisfaction has been established.

Our final obligation is the non-interference proof. Trivially, assertions in the producer are not interfered with by any operation in the consumer since the consumer never alters any variable used in any producer assertion. We claim that assertions in the consumer are also interference-free. The producer never alters any explicit variable used in a consumer assertion. The **send** statements in the producer do alter the implicit variables of I, and I is a conjunct in every consumer assertion. We must prove that, for every assertion in the consumer, execution of the **send** statements **s1** through **s8** in the producer do not invalidate that assertion. As an illustrative example, consider the loop invariant in the consumer and the **send** labeled s4 in the producer. Our non-interference obligation is to show

$$\{\text{done} = \mathit{false} \wedge \rho = \mathcal{C}(\text{X},\text{C},\text{D}) \wedge I \wedge \text{pre(s4)}\}\ \text{s4}\ \{\text{done} = \mathit{false} \wedge \rho = \mathcal{C}(\text{X},\text{C},\text{D}) \wedge I\}$$

or equivalently,

$$\{\text{done} = \mathit{false} \wedge \rho = \mathcal{C}(\text{X},\text{C},\text{D}) \wedge I \wedge \text{i} = \text{a} \wedge \text{a} \leq \text{M} \wedge I\}\ \text{s4}\ \{\text{done} = \mathit{false} \wedge \rho = \mathcal{C}(\text{X},\text{C},\text{D}) \wedge I\}$$

The truth of $\rho = \mathcal{C}(\text{X},\text{C},\text{D})$ cannot be affected by s4 since the **send** alters none of the variables, explicit or implicit, appearing in that predicate. Our argument concerning the invariant truth of I across s4 in the proof in isolation of the producer applies here as well. Hence, s4 cannot interfere with the loop invariant of the consumer. An analogous argument applies for every assertion in the consumer. Furthermore, a similar line of reasoning allows us to conclude that s1 through s3 and s5 through s8, the other **send** statements in the producer, do not interfere with any assertion in the consumer.

3.6 Verification of Flush Batching Applications

As in Section 3.5, the following three examples apply the axiomatic proof methodology of Section 3.4. The complexity of the verification process is reduced, however, as only two message types are transmitted in each example. Groups of ordinary messages are intended to convey information from a "producer" process to a "consumer" process. Flush messages batch the groups of ordinary messages in a manner particular to each type of flush. (Recall Section 2.8.2.) In each example, the messages contain two data fields: *batch* and *num*. For an ordinary message, *batch* is the batch to which the message belongs; *num* is the number of the message within its batch. In a flush message, *batch* denotes the batch which the message is delimiting; *num* is always zero.

In all three examples, the following program variables convey identical information [CKA93]:

In the Producer:

bat	: integer	;	current batch number
job	: integer	;	current message number within a batch
mb	: integer array	;	mb[i] is number of messages in batch i

In the Consumer:

jobs	: set	;	set containing ordinary messages received
cb	: integer	;	current batch number

We assume that each element of array mb is a positive integer, that mb is defined externally to the processes, but that it is known to both the producer and the consumer.

3.6.1 Batch Example 1: Illustrating Proof Rules for ORD/2F

In the first batch example, only ordinary and two-way flush messages are transmitted. Figure 3.3 shows the covering relation for this example.

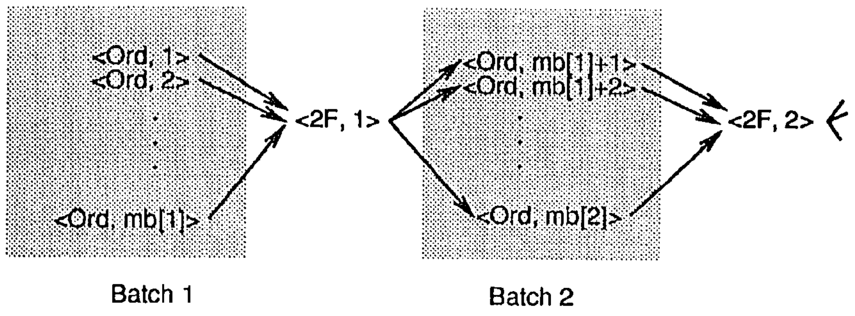

Figure 3.3: Batches Delimited With Two-way Flushes

We define the invariant, as in Section 3.5,

$$I = \forall m \in \sigma_F : \mathcal{I}(m)$$

in order to describe the state of the implicit variables at the transmission of message m. As shown in the figure, m is the composite

$$m = <m.type, m.batch, m.num>,$$

where $m.type$ is the message type, and $m.batch$ and $m.num$ are as described in the preceding discussion.

In this batch example, the tedium of $\mathcal{I}(m)$ is greatly reduced:

$$\mathcal{I}(m) \equiv \mathcal{I}_{\sigma(m)} \wedge \mathcal{I}_{\prec(m)} \wedge \mathcal{I}_{\tau(m)} \wedge \mathcal{I}_{\Omega(m)},$$

where

$$\begin{aligned}
\mathcal{I}_{\sigma(m)} \equiv \sigma(m) &= \{<\mathbf{2F}, i, 0>|0 < i \leq m.batch\} \oplus \{<\mathbf{Ord}, i, j>|0 < i < m.batch \wedge 0 < j \leq \mathrm{mb}[i]\} \\
&\oplus\{<\mathbf{Ord}, m.batch, i>|0 < i \leq m.num)\}; \\
\mathcal{I}_{\prec(m)} \equiv \prec(m) &= \{(<\mathbf{2F}, i, 0>, <\mathbf{Ord}, i, j>)|0 < i < m.batch \wedge 0 < j \leq \mathrm{mb}[i]\} \\
&\oplus\{(<\mathbf{Ord}, i, j>, <\mathbf{2F}, i+1, 0>)|0 < i < m.batch \wedge 0 < j \leq \mathrm{mb}[i]\} \\
&\oplus\{(<\mathbf{2F}, m.batch, 0>, <\mathbf{Ord}, m.batch, i>)|0 < i \leq m.num\}; \\
\mathcal{I}_{\tau(m)} \equiv \tau(m) &= \{<\mathbf{2F}, m.batch, 0>\}; \\
\mathcal{I}_{\Omega(m)} \equiv \Omega(m) &= \{<\mathbf{2F}, m.batch, 0>|m.num = 0\} \oplus \{<\mathbf{Ord}, m.batch, i>|0 < i \leq m.num\}.
\end{aligned}$$

The reduction to $\mathcal{I}(m)$, comparing to $\mathcal{I}(m)$ in Section 3.5, is due to the simpler delivery order. Compare the covering relation of Figure 3.2 with that of Figure 3.3. Since the arrows, representing delivery order requirements, are more predictable in the second figure, it is easier to describe the state of the implicit variables at any given time.

```
PRODUCER::
                    { $I$ }
bat, job := 0, 0;
                    { bat,job = 0,0 $\wedge$ $I$ }
While true do
                    { job = mb[bat] $\wedge$ $I$ }
    bat, job := bat + 1, 0;
                    { bat > 0 $\wedge$ job = 0 $\wedge$ $I$ }
    s1: send (2F, bat, job) on F;
                    { bat > 0 $\wedge$ job = 0 $\wedge$ $I$ }
    While job < mb[bat] do
                    { job < mb[bat] $\wedge$ $I$ }
```

```
                job := job + 1;
                    { $job \le mb[bat] \wedge I$ }
                s2: send (Ord, bat, job) on F;
                    { $job \le mb[bat] \wedge I$ }
            od;
                    { $job = mb[bat] \wedge I$ }
        od;
```

We omit discussion of the proof in isolation of the producer process as it is straightforward. Similar to Section 3.5, the invariant truth of I in the producer can be inductively validated as successive messages are transmitted.

We define the set of messages which have been consumed thus far, in terms of the consumer's variables cb and jobs:

$$\mathcal{C}(\text{cb}, \text{jobs}) \equiv \{\mu \mid (\mu = <\textbf{Ord}, x, y> \wedge <x, y> \in \text{jobs}) \vee (\mu = <\textbf{2F}, z, 0> \wedge 0 < z \le \text{cb})\}.$$

A newly received message (with its type assigned to the consumer's variable mtype, its batch assigned to mbatch, and its number assigned to mnum) must satisfy the following parametric assertion:

$$\mathcal{N}(\text{cb}, \text{jobs}) \equiv$$

$$(\text{mtype} = \textbf{Ord} \wedge <\text{mbatch}, \text{mnum}> \notin \text{jobs} \wedge \text{mbatch} = \text{cb} \wedge 0 < \text{mnum} \le \text{mb}[\text{mbatch}])$$
$$\vee(\text{mtype} = \textbf{2F} \wedge \text{mbatch} = \text{cb} + 1 \wedge \text{mnum} = 0 \wedge |\text{jobs}| = \sum_{i=1}^{\text{cb}} \text{mb}[i])$$

As before, establishing the truth of $\mathcal{N}(\text{cb}, \text{jobs})$ is the primary task of the satisfaction proof. The annotated consumer process appears as:

```
CONSUMER::
            { $\rho = \emptyset \wedge I$ }
jobs, cb := $\emptyset$, 0;
            { $jobs = \emptyset \wedge cb = 0 \wedge \rho = \mathcal{C}(cb, jobs) \wedge I$ }
While true do
            { $\rho = \mathcal{C}(cb, jobs) \wedge I$ }
    r1: receive (mtype, mbatch, mnum) from F;
```

```
            { $\mathcal{N}(\text{cb},\text{jobs}) \wedge \rho = \mathcal{C}(\text{cb},\text{jobs}) \oplus \{\text{<mtype, mbatch, mnum>}\} \wedge I$ }
        Case mtype of
            Ord:  jobs := jobs ∪ {<mbatch, mnum>};
{$\text{mtype} = \textbf{Ord} \wedge \text{<mbatch, mnum>} \in \text{jobs} \wedge \text{mbatch} = \text{cb} \wedge 0 < \text{mnum} \leq \text{mb[mbatch]} \wedge \rho = \mathcal{C}(\text{cb},\text{jobs}) \wedge I$ }
            2F:    cb := cb + 1;
    { $\text{mtype} = \textbf{2F} \wedge \text{mbatch} = \text{cb} \wedge \text{mnum} = 0 \wedge |\text{jobs}| = \sum_{i=1}^{\text{cb}-1} \text{mb}[i] \wedge \rho = \mathcal{C}(\text{cb},\text{jobs}) \wedge I$ }
        esac;
            { $\rho = \mathcal{C}(\text{cb},\text{jobs}) \wedge I$ }
    od;
```

Let us now justify the miraculous postcondition of the **receive**. The satisfaction proof below is rather detailed; the satisfaction proofs for the next two batch examples contain less detail since all three are similar in form. Let $MTEXT$ denote any message eligible to be received, i.e., assigned to <mtype, mbatch, mnum>. Our obligation in the satisfaction proof is to justify the following implication:

$$\text{pre}(\mathbf{r1}) \wedge (MTEXT \in \sigma \ominus \rho) \wedge (\forall m : m \in \sigma \wedge m \not\prec MTEXT :: m \in \rho)$$
$$\Rightarrow \text{post}(\mathbf{r1})^{\text{<mtype,mbatch,mnum>},\rho}_{MTEXT,\qquad \rho\oplus\{MTEXT\}},$$

or equivalently, we need to show

$$(\rho = \mathcal{C}(\text{cb},\text{jobs}) \wedge I) \wedge (MTEXT \in \sigma \ominus \rho) \wedge (\forall m : m \in \sigma \wedge m \not\prec MTEXT :: m \in \rho)$$
$$\Rightarrow (\mathcal{N}(\text{cb},\text{jobs}) \wedge \rho = \mathcal{C}(\text{cb},\text{jobs})$$
$$\oplus\{\text{<mtype, mbatch, mnum>}\} \wedge I)^{\text{<mtype,mbatch,mnum>},\rho}_{MTEXT,\qquad \rho\oplus\{MTEXT\}}.$$

Given that $(\rho = \mathcal{C}(\text{cb},\text{jobs})) \wedge I$ is in the antecedent, then clearly

$$(\rho \oplus \{MTEXT\} = \mathcal{C}(\text{cb},\text{jobs}) \oplus \{MTEXT\}) \wedge I.$$

Therefore, we concentrate on establishing $\mathcal{N}(\text{cb},\text{jobs})$.

We consider $MTEXT$, the message being added to ρ, when the **receive** statement executes. The field mtype must be **Ord** or **2F**; this follows from $\mathcal{I}_{\sigma(MTEXT)}$. For each message type, **Ord** or **2F**, we deduce three clauses in order to establish $\mathcal{N}(\text{cb},\text{jobs})$. For a two-way flush, we verify the assertion in complete detail. We follow this complete proof with a verbal description that justifies the receipt of an ordinary message. (The complete proof for the receipt of an ordinary message is similar to the two-way flush case.) To begin, let us assume the message being received is a two-way flush.

Statement	Justification
A1. $MTEXT = <\mathbf{2F}, f, g>$	Assumption
A2. $MTEXT \in \sigma$	Antecedent ($MTEXT \in \sigma \ominus \rho$)
A3. $<\mathbf{2F}, \text{cb}, 0> \in \rho \wedge \nexists$ $<\mathbf{2F}, k, 0> \in \rho$ for $k >$ cb	Antecedent ($\rho = \mathcal{C}(\text{cb},\text{jobs})$)
A4. $f = \text{cb} + 1$	A2, A3, and Antecedent ($(\forall m : m \in \sigma \wedge m \not\prec MTEXT \Rightarrow m \in \rho)$ and $\mathcal{I}_{\prec(MTEXT)}$)
A5. $g = 0$	A2 and Antecedent ($\mathcal{I}_{\sigma(MTEXT)}$)
A6. $\forall m : m \in \sigma(MTEXT) \Rightarrow m \in \rho$	A2 and Antecedent ($(\forall m : m \in \sigma \wedge m \not\prec MTEXT \Rightarrow m \in \rho)$ and $\mathcal{I}_{\prec(MTEXT)}$)
A7. $\forall m : m \in \sigma(MTEXT) \Rightarrow m$ is unique	A2 and Antecedent ($\mathcal{I}_{\sigma(MTEXT)}$)
A8. $\forall m : m = <\mathbf{Ord}, c, d> \wedge m \in \rho \Rightarrow <c, d> \in \text{jobs}$	Antecedent ($\rho = \mathcal{C}(\text{cb},\text{jobs})$)
A9. $\lvert\text{jobs}\rvert = \sum_{i=1}^{f-1} \text{mb}[i]$	A2, A6, A7, A8, and Antecedent ($\mathcal{I}_{\sigma(MTEXT)}$)
A10. $\lvert\text{jobs}\rvert = \sum_{i=1}^{\text{cb}} \text{mb}[i]$	A4

A4, A5, and A10 establish half of $\mathcal{N}(\text{cb},\text{jobs})$, i.e., when $MTEXT$ is a two-way flush. If we assume the message being received is an ordinary message, $MTEXT = <\mathbf{Ord}, c, d>$, we show three facts about this message as well. From the antecedent (($\rho = \mathcal{C}(\text{cb},\text{jobs})$) and ($MTEXT \in \sigma \ominus \rho$)) and the fact that each $<c, d>$ is unique, the composite $<c, d>$ has not been received before, and thus, cannot be an element of jobs. We also know, since $<\mathbf{2F}, \text{cb}, 0>$ is the last two-way flush received, that c must be equal to the current batch (cb). Furthermore, from $\mathcal{I}_{\sigma(MTEXT)}$, we know the number of this ordinary message must be within the current batch.

Combining what we have established,

$$
\begin{aligned}
&((MTEXT = <\mathbf{Ord}, c, d> \wedge <c, d> \not\in \text{jobs} \wedge c = \text{cb} \wedge 0 < d \leq \text{mb}[c]) \\
&\vee(MTEXT = <\mathbf{2F}, f, g> \wedge f = \text{cb} + 1 \wedge g = 0 \wedge |\text{jobs}| = \textstyle\sum_{i=1}^{\text{cb}} \text{mb}[i])) \\
&\wedge(\rho \oplus \{MTEXT\} = \mathcal{C}(\text{cb}, \text{jobs}) \oplus \{MTEXT\} \wedge I) \\
\equiv& \\
&(((<\text{mtype}, \text{mbatch}, \text{mnum}> = <\mathbf{Ord}, c, d> \wedge <c, d> \not\in \text{jobs} \wedge c = \text{cb} \wedge 0 < d \leq \text{mb}[c]) \\
&\vee(<\text{mtype}, \text{mbatch}, \text{mnum}> = <\mathbf{2F}, f, g> \wedge f = \text{cb} + 1 \wedge g = 0 \wedge |\text{jobs}| = \textstyle\sum_{i=1}^{\text{cb}} \text{mb}[i])) \\
&\wedge(\rho = \mathcal{C}(\text{cb}, \text{jobs}) \oplus \{<\text{mtype}, \text{mbatch}, \text{mnum}>\} \wedge I))^{<\text{mtype},\text{mbatch},\text{mnum}>,\ \rho}_{MTEXT,\ \rho\oplus\{MTEXT\}} \\
\equiv& \\
&(\mathcal{N}(\text{cb}, \text{jobs}) \wedge \rho = \mathcal{C}(\text{cb}, \text{jobs}) \oplus \{<\text{mtype}, \text{mbatch}, \text{mnum}>\} \wedge I)^{<\text{mtype},\text{mbatch},\text{mnum}>,\ \rho}_{MTEXT,\ \rho\oplus\{MTEXT\}}.
\end{aligned}
$$

The above result is the required consequent, and hence, satisfaction has been established.

To prove non-interference, we direct the reader to the reasoning for non-interference in Section 3.5. Since the two proofs are similar, we omit it in this example.

3.7 Batch Example 2: Illustrating Proof Rules for ORD/BF

In this second batch example, each batch is preceded by a backward flush. Figure 3.4 depicts this message-passing scenario. Again, we define the invariant, $I = \forall m \in \sigma : \mathcal{I}(m)$; as in Example 3.6.1, we let m denote the composite $<m.type, m.batch, m.num>$. We need to make only minor changes to $\mathcal{I}(m)$ in the first batch example in order to describe the

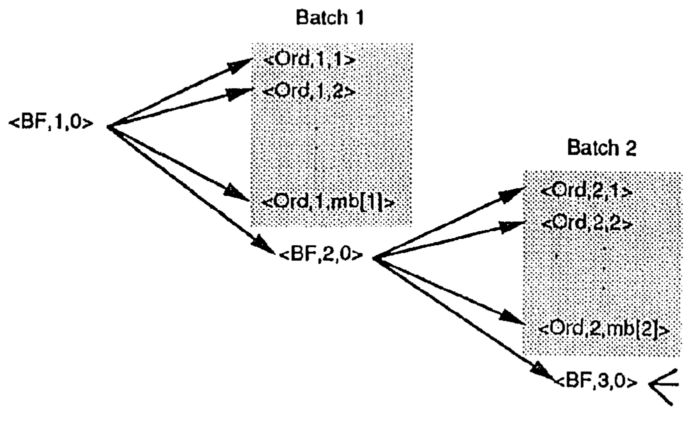

Figure 3.4: Batches Preceded by Backward Flushes

transmission of message m in the new batching system:

$$\mathcal{I}(m) \equiv \mathcal{I}_{\sigma(m)} \wedge \mathcal{I}_{\prec(m)} \wedge \mathcal{I}_{\tau(m)} \wedge \mathcal{I}_{\Omega(m)},$$

where

$$\begin{aligned}
\mathcal{I}_{\sigma(m)} \equiv \sigma(m) = & \{<\mathbf{BF}, i, 0> \mid 0 < i \leq m.batch\} \\
& \oplus\{<\mathbf{Ord}, i, j> \mid 0 < i < m.batch \wedge 0 < j \leq \mathrm{mb}[i]\} \\
& \oplus\{<\mathbf{Ord}, m.batch, i> \mid 0 < i \leq m.num\}; \\
\mathcal{I}_{\prec(m)} \equiv \prec(m) = & \{(<\mathbf{BF}, i, 0>, <\mathbf{Ord}, i, j>) \mid 0 < i < m.batch \wedge 0 < j \leq \mathrm{mb}[i]\} \\
& \oplus\{(<\mathbf{BF}, i, 0>, <\mathbf{BF}, i+1, 0>) \mid 0 < i < m.batch\} \\
& \oplus\{(<\mathbf{BF}, m.batch, 0>, <\mathbf{Ord}, m.batch, i>) \mid 0 < i \leq m.num\}; \\
\mathcal{I}_{\tau(m)} \equiv \tau(m) = & \{<\mathbf{BF}, m.batch, 0\}; \\
\mathcal{I}_{\Omega(m)} \equiv \Omega(m) = & \{<\mathbf{BF}, m.batch, 0> \mid m.num = 0\} \\
& \oplus\{<\mathbf{Ord}, i, j> \mid 0 < i < m.batch \wedge 0 < j \leq \mathrm{mb}[i]\} \\
& \oplus\{<\mathbf{Ord}, m.batch, i> \mid 0 < i \leq m.num\}.
\end{aligned}$$

The assertions in the producer proof, detailed below for the sake of completeness, are the same as the ones in the previous example. The only change in the producer is that the **send** at label **s1** transmits a backward flush rather than a two-way flush. Similar to the example in Section 3.5, the invariant truth of I in the producer is inductively validated as successive messages are sent.

```
PRODUCER::
                { $I$ }
bat, job := 0, 0;
                { bat,job = 0,0 $\wedge$ $I$ }
While true do
                { job = mb[bat] $\wedge$ $I$ }
    bat, job := bat + 1, 0;
                { bat > 0 $\wedge$ job = 0 $\wedge$ $I$ }
    s1: send (BF, bat, job) on $F$;
                { bat > 0 $\wedge$ job = 0 $\wedge$ $I$ }
    While job < mb[bat] do
                { job < mb[bat] $\wedge$ $I$ }
        job := job + 1;
                { job $\leq$ mb[bat] $\wedge$ $I$ }
        s2: send (Ord, bat, job) on $F$;
                { job $\leq$ mb[bat] $\wedge$ $I$ }
```

```
            od;
                          { $job = mb[bat] \wedge I$ }
        od;
```

The definition of $\mathcal{C}$, the set of messages consumed thus far, is similar to that of the previous example:

$$\mathcal{C}(\text{cb},\text{jobs}) = \{\mu \mid (\mu = <\textbf{Ord}, x, y> \wedge <x, y> \in \text{jobs}) \vee (\mu = <\textbf{BF}, z, 0> \wedge 0 < z \leq \text{cb})\}.$$

The parametric assertion $\mathcal{N}$, which describes a newly received message, must be modified according to the following reasoning. As in Example 3.6.1, the newly received message has its type assigned to the consumer's variable "mtype", its batch assigned to "mbatch", and its number assigned to "mnum". If the message currently received is $<\textbf{Ord}, c, d>$, then c must be less than or equal to the most current batch announced by a backward flush. If the message currently received is a backward flush, then the number of jobs received thus far must be less than or equal to the sum of the sizes of the batches previously announced. Equivalently,

$$\mathcal{N}(\text{cb},\text{jobs}) \equiv$$

$$(\text{mtype} = \textbf{Ord} \wedge <\text{mbatch}, \text{mdata}> \notin \text{jobs} \wedge \text{mbatch} \leq \text{cb} \wedge 0 < \text{mnum} \leq \text{mb}[\text{mbatch}])$$
$$\vee(\text{mtype} = \textbf{BF} \wedge \text{mbatch} = \text{cb} + 1 \wedge \text{mnum} = 0 \wedge |\text{jobs}| \leq \sum_{i=1}^{\text{cb}} \text{mb}[i]).$$

The annotated consumer process appears as:

```
CONSUMER::
            { $\rho = \emptyset \wedge I$ }
jobs, cb := $\emptyset$, 0;
            { $jobs = \emptyset \wedge cb = 0 \wedge \rho = \mathcal{C}(cb, jobs) \wedge I$ }
While true do
            { $\rho = \mathcal{C}(cb, jobs) \wedge I$ }
    r1: receive (mtype, mbatch, mnum) from $F$;
            { $\mathcal{N}(cb, jobs) \wedge \rho = \mathcal{C}(cb, jobs) \oplus \{<mtype, mbatch, mnum>\} \wedge I$ }
    Case mtype of
        Ord:  jobs := jobs $\cup$ {<mbatch, mnum>};
```

```
{ $\mathrm{mtype} = \mathrm{Ord} \wedge <\mathrm{mbatch}, \mathrm{mnum}> \in \mathrm{jobs} \wedge \mathrm{mbatch} \leq \mathrm{cb} \wedge 0 < \mathrm{mnum} \leq \mathrm{mb}[\mathrm{mbatch}] \wedge \rho = \mathcal{C}(\mathrm{cb}, \mathrm{jobs}) \wedge I$ }
            BF:    cb := cb + 1;
        { $\mathrm{mtype} = \mathrm{BF} \wedge \mathrm{mbatch} = \mathrm{cb} \wedge \mathrm{mnum} = 0 \wedge |\mathrm{jobs}| \leq \sum_{i=1}^{\mathrm{cb}-1} \mathrm{mb}[i] \wedge \rho = \mathcal{C}(\mathrm{cb}, \mathrm{jobs}) \wedge I$ }
        esac;
            { $\rho = \mathcal{C}(\mathrm{cb}, \mathrm{jobs}) \wedge I$ }
    od;
```

The satisfaction proof is outlined below. The newly received message, $MTEXT$, must be an ordinary or a backward flush message. The antecedent of the satisfaction rule is

$$(\rho = \mathcal{C}(\mathrm{cb}, \mathrm{jobs}) \wedge I) \wedge (MTEXT \in \sigma \ominus \rho) \wedge (\forall m : m \in \sigma \wedge m \not\prec MTEXT \Rightarrow m \in \rho).$$

This leads to the desired consequent,

$$\begin{aligned}
&((MTEXT = <\mathrm{Ord}, c, d> \wedge <c, d> \notin \mathrm{jobs} \wedge c \leq \mathrm{cb} \wedge 0 < d \leq \mathrm{mb}[c]) \\
&\quad \vee (MTEXT = <\mathrm{BF}, f, g> \wedge f = \mathrm{cb} + 1 \wedge g = 0 \wedge |\mathrm{jobs}| \leq \textstyle\sum_{i=1}^{\mathrm{cb}} \mathrm{mb}[i])) \\
&\quad \wedge(\rho \oplus \{MTEXT\} = \mathcal{C}(\mathrm{cb}, \mathrm{jobs}) \oplus \{MTEXT\} \wedge I) \\
&\equiv \\
&(\mathcal{N}(\mathrm{cb}, \mathrm{jobs}) \wedge \rho = \mathcal{C}(\mathrm{cb}, \mathrm{jobs}) \oplus \{<\mathrm{mtype}, \mathrm{mbatch}, \mathrm{mnum}>\} \wedge I)^{<\mathrm{mtype}, \mathrm{mbatch}, \mathrm{mnum}>, \rho}_{MTEXT, \quad \rho \oplus \{MTEXT\}}.
\end{aligned}$$

Non-interference follows from a similar line of reasoning in the example of Section 3.5.

3.8 Batch Example 3: Illustrating Proof Rules for ORD/FF

If we let forward flushes terminate batches, Figure 3.5, we get a totally different effect from that of the previous two batch examples.

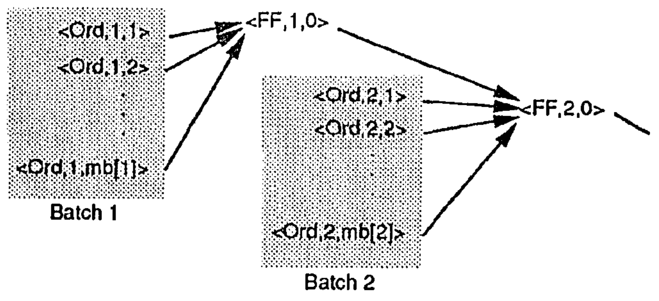

Figure 3.5: Batches Terminated with Forward Flushes

In this third and final batch example, the invariant which describes the implicit variables must be substantially modified to deal with the forward flush. The producer algorithm is structurally different from the previous two producers; in this example it is necessary to transmit the forward flush *after* the ordinary messages in the batch. As before, define $I = \forall m \in \sigma : \mathcal{I}(m)$, where

$$\mathcal{I}(m) \equiv \mathcal{I}_{\sigma(m)} \wedge \mathcal{I}_{\prec(m)} \wedge \mathcal{I}_{\tau(m)} \wedge \mathcal{I}_{\Omega(m)}.$$

Elaborating each conjunct:

$$\begin{aligned}
\mathcal{I}_{\sigma(m)} \equiv \sigma(m) = & \{<\mathbf{Ord}, i, j>|0 < i < m.batch \wedge 0 < j \leq \mathrm{mb}[i]\} \oplus \{<\mathbf{FF}, i, 0>|0 < i < m.batch\} \\
& \oplus\{<\mathbf{Ord}, m.batch, i>|0 < i \leq m.num\} \oplus \{<\mathbf{FF}, m.batch, 0>|m.num = 0\} \\
& \oplus\{<\mathbf{Ord}, m.batch, i>|m.num = 0 \wedge 0 < i \leq \mathrm{mb}[m.batch]\}; \\
\mathcal{I}_{\prec(m)} \equiv \prec(m) = & \{(<\mathbf{Ord}, i, j>, <\mathbf{FF}, i, 0>)|0 < i < m.batch \wedge 0 < j \leq \mathrm{mb}[i]\} \\
& \oplus\{(<\mathbf{FF}, i-1, 0>, <\mathbf{FF}, i, 0>)|1 < i < m.batch\} \\
& \oplus\{(<\mathbf{Ord}, m.batch, i>, <\mathbf{FF}, m.batch, 0>)|m.num = 0 \wedge 0 < i \leq \mathrm{mb}[m.batch]\} \\
& \oplus\{(<\mathbf{FF}, i-1, 0>, <\mathbf{FF}, i, 0>)|m.num = 0 \wedge 1 < m.batch = i\}; \\
\mathcal{I}_{\tau(m)} \equiv \tau(m) = & \emptyset; \\
\mathcal{I}_{\Omega(m)} \equiv \Omega(m) = & \{<\mathbf{Ord}, m.batch, i>|0 < i \leq m.num\} \\
& \oplus\{<\mathbf{FF}, i, 0>|m.num \neq 0 \wedge 0 < i = m.batch - 1\} \\
& \oplus\{<\mathbf{FF}, m.batch, 0>|m.num = 0\}.
\end{aligned}$$

Although the producer algorithm is changed from the previous two producers, the proof in isolation continues to be straightforward. Moreover, the invariant truth of I in this producer is established using the same reasoning of the previous examples.

```
PRODUCER::
                { $I$ }
bat, job := 0, 0;
                { $\mathrm{bat}, \mathrm{job} = 0, 0 \wedge I$ }
While true do
                { $\mathrm{job} = \mathrm{mb}[\mathrm{bat}] \wedge I$ }
    bat, job := bat + 1, 0;
                { $\mathrm{bat} > 0 \wedge \mathrm{job} = 0 \wedge I$ }
    While job < mb[bat] do
                { $\mathrm{job} < \mathrm{mb}[\mathrm{bat}] \wedge I$ }
        job := job + 1;
```

```
                { $job \le mb[bat] \wedge I$ }
            s2: send (Ord, bat, job) on $F$;
                { $job \le mb[bat] \wedge I$ }
        od;
                { $job = mb[bat] \wedge I$ }
        s1: send (FF, bat, 0) on $F$;
                { $job = mb[bat] \wedge I$ }
    od;
```

For the consumer, the parametric description of the set of consumed messages, $\mathcal{C}$, is essentially the same as previously defined:

$$\mathcal{C}(\text{cb},\text{jobs}) = \{\mu \mid (\mu = <\textbf{Ord}, x, y> \wedge <x,y> \in \text{jobs}) \vee (\mu = <\textbf{FF}, z, 0> \wedge 0 < z \le \text{cb})\}.$$

The assertion $\mathcal{N}$, describing a newly received message, is somewhat changed in keeping with the structure of the application. It is no longer necessary that c, from a newly received $<\textbf{Ord}, c, d>$, be less than or equal to the current batch. Instead, c must only be larger than the last batch number terminated by a forward flush. In addition, if the new message received is a forward flush, then the number of jobs received is at least the sum of the number of jobs in all of the currently terminated batches.

$$\mathcal{N}(\text{cb},\text{jobs}) \equiv$$

$$(\text{mtype} = \textbf{Ord} \wedge <\text{mbatch},\text{mnum}> \notin \text{jobs} \wedge \text{mbatch} > \text{cb} \wedge 0 < \text{mnum} \le \text{mb}[\text{mbatch}])$$
$$\vee(\text{mtype} = \textbf{FF} \wedge \text{mbatch} = \text{cb} + 1 \wedge \text{mnum} = 0 \wedge \sum_{i=1}^{\text{cb}+1} \text{mb}[i] \le |\text{jobs}|).$$

```
CONSUMER::
            { $\rho = \emptyset \wedge I$ }
jobs, cb := $\emptyset$, 0;
            { $jobs = \emptyset \wedge cb = 0 \wedge \rho = \mathcal{C}(cb, jobs) \wedge I$ }
While true do
            { $\rho = \mathcal{C}(cb, jobs) \wedge I$ }
    r1: receive (mtype, mbatch, mnum) from $F$;
```

```
                { $\mathcal{N}(\text{cb},\text{jobs}) \wedge \rho = \mathcal{C}(\text{cb},\text{jobs}) \oplus \{<\text{mtype},\text{mbatch},\text{mnum}>\} \wedge I$ }
            Case mtype of
                Ord:   jobs := jobs ∪ {<mbatch, mnum>};
{$\text{mtype} = \textbf{Ord} \wedge <\text{mbatch},\text{mnum}> \in \text{jobs} \wedge \text{mbatch} > \text{cb} \wedge 0 < \text{mnum} \leq \text{mb}[\text{mbatch}] \wedge \rho = \mathcal{C}(\text{cb},\text{jobs}) \wedge I$ }
                FF:    cb := cb + 1;
       { $\text{mtype} = \textbf{FF} \wedge \text{mbatch} = \text{cb} \wedge \text{mnum} = 0 \wedge \sum_{i=1}^{\text{cb}} \text{mb}[i] \leq |\text{jobs}| \wedge \rho = \mathcal{C}(\text{cb},\text{jobs}) \wedge I$ }
            esac;
                { $\rho = \mathcal{C}(\text{cb},\text{jobs}) \wedge I$ }
        od;
```

The satisfaction proof is shown below. As before, we assume

$$(\rho = \mathcal{C}(\text{cb},\text{jobs}) \wedge I) \wedge (MTEXT \in \sigma \ominus \rho) \wedge (\forall m : m \in \sigma \wedge m \not\prec MTEXT \Rightarrow m \in \rho).$$

We then show, using I and the F-channel network axiom,

$$\begin{aligned}
&((MTEXT = <\textbf{Ord}, c, d> \wedge <c, d> \not\in \text{jobs} \wedge c > \text{cb} \wedge 0 < d \leq \text{mb}[c]) \\
&\ \vee (MTEXT = <\textbf{FF}, f, g> \wedge f = \text{cb} + 1 \wedge g = 0 \wedge \textstyle\sum_{i=1}^{\text{cb}+1} \text{mb}[i] \leq |\text{jobs}|)) \\
&\ \wedge(\rho \oplus \{MTEXT\} = \mathcal{C}(\text{cb},\text{jobs}) \oplus \{MTEXT\} \wedge I) \\
\equiv& \\
&(\mathcal{N}(\text{cb},\text{jobs}) \wedge \rho = \mathcal{C}(\text{cb},\text{jobs}) \oplus \{<\text{mtype},\text{mbatch},\text{mnum}>\} \wedge I)^{<\text{mtype},\text{mbatch},\text{mnum}>,\ \rho}_{MTEXT,\ \ \rho \oplus \{MTEXT\}}.
\end{aligned}$$

Once again, we rely on the reasoning for non-interference of the example in Section 3.5 to verify that this system is interference-free.

3.9 Soundness and Completeness

In this section, we show that the axiomatic proof methodology for F-channels is sound and relatively complete. To show soundness, we illustrate that what we prove in our verification methodology is true. Relative completeness, relative to some complete deductive system, establishes that the methodology can prove anything that is true. There is no complete deductive system for natural numbers, therefore, no programming language which uses the natural numbers is complete. To avoid this issue, as suggested by Cook [Coo78], we assume we have a complete deductive system; we then illustrate that our proof system can prove anything which is true. Since we know that many axiomatic proof methodologies for

CSP are sound and relatively complete [AFR80, LG81, Sou84], we prove soundness and relative completeness for our proof system by simulating an F-channel with CSP sends and receives. We then derive the F-channel axiomatic proof system from the CSP code using a proof system for CSP which has been shown to be sound and relatively complete.

Theorem 11 *The axiomatic proof methodology for processes communicating with F-channels is sound and relatively complete.*

Proof: Suppose we have a distributed program that transmits flush messages on F-channel F from process S to process R. To model the F-channel, CSP processes S and R synchronously communicate with CSP process B. The main function of B is to accept messages from S and to transmit these messages to R in an order that is consistent with the definition of an F-channel. σ_F, $\prec_F$, τ_F, and Ω_F are variables that model the F-channel communication state. A(m), B(m), and C(m) are functions, described in Section 3.4, that modify these variables.

The F-channel communication state is maintained by process B. By definition of an F-channel, a message cannot be delivered before it is transmitted. In addition, a message must be delivered in the order denoted by $\prec\!\!\!+_F$. These two properties are stated in the Flush Channel Network Axiom. In our simulation of an F-channel, we define the invariant I_{FC} to be consistent with the network axiom.

$$\begin{aligned} \mathrm{I}_{FC}: \quad & \rho_F \subseteq \sigma_F, \\ & \text{For } m, m' \in \sigma_F, m' \in \rho_F \Rightarrow (\forall m : m \prec\!\!\!+_F m' :: m \in \rho_F). \end{aligned}$$

I_{FC} is an implicit conjunct in every assertion of the CSP code. The following annotated CSP code implements B.

```
B :: do  $r_b$: $S?(\sigma_F, \prec_F, \Omega_F, \tau_F) \rightarrow$ skip;
       []
         $s_b$: not(empty($\sigma_F \ominus \rho_F$)) $\rightarrow$
                      { $(\sigma_F \ominus \rho_F) \neq \emptyset$ }
                  foreach $\mu \in (\sigma_F \ominus \rho_F)$ do
                      { $\mu \in (\sigma_F \ominus \rho_F)$ }
                      if $Pred(\mu) \subseteq \rho_F \rightarrow R!(<\mu.type, \mu.data>, \rho_F \oplus \{\mu\})$; break;
                      []
                        true $\rightarrow$ skip;
                      fi;
                  od;
    od;
```

B nondeterministically chooses one of two actions: it may accept an incoming message from S; or it may, if an eligible message exists, transmit a message to R.

Send Axioms: We consider each of the four flush message types separately. To implement the transmission of an ordinary message,

$$S_1\text{: } \mathbf{send}\ <\mathbf{Ord}, \text{data}>\ \mathbf{on}\ F,$$

we simulate the communication statement with the following CSP transmission.

$$s_1\text{: } B!(\sigma_F \oplus \{<\mathbf{Ord}, \text{data}>\}, \prec_F \oplus A(<\mathbf{Ord}, \text{data}>),$$
$$\Omega_F \oplus \{<\mathbf{Ord}, \text{data}>\} \ominus B(<\mathbf{Ord}, \text{data}>), \tau_F).$$

To transmit a two-way flush message,

$$S_2\text{: } \mathbf{send}\ <\mathbf{2F}, \text{data}>\ \mathbf{on}\ F,$$

we execute the following synchronous communication.

$$s_2\text{: } B!(\sigma_F \oplus \{<\mathbf{2F}, \text{data}>\}, \prec_F \oplus C(<\mathbf{2F}, \text{data}>),$$
$$\{<\mathbf{2F}, \text{data}>\}, \{<\mathbf{2F}, \text{data}>\}).$$

We implement the transmission of a forward flush message,

$$S_3\text{: } \textbf{send <FF,data> on } F,$$

with the following CSP transmission:

$$s_3\text{: } B!(\sigma_F \oplus \{\text{<FF, data>}\}, \prec_F \oplus C(\text{<FF, data>}),$$
$$\{\text{<FF, data>}\}, \tau_F).$$

A backward flush message transmission,

$$S_4\text{: } \textbf{send <BF,data> on } F,$$

is simulated by

$$s_4\text{: } B!(\sigma_F \oplus \{\text{<BF, data>}\}, \prec_F \oplus A(\text{<BF, data>}),$$
$$\Omega_F \oplus \{\text{<BF, data>}\} \ominus B(\text{<BF, data>}), \{\text{<Ord, data>}\}).$$

Regardless of the message type, the flush communication statement is modeled by a synchronous transmission to B. We can derive the F-channel send axioms from the satisfaction of the four synchronous communication statements. For example, consider the transmission of a backward flush message. Suppose m is the message <BF, data>. For s_4, the simulated F-channel communication statement, we know that

$$\{T\} \quad s_4 \quad \{W\}$$

from the assertions surrounding a CSP transmission statement. To establish satisfaction for the communication between S (at s_4) and R (at r_b), we use the Synchronous Satisfaction Rule.

$$(T) \Rightarrow (W)^{\sigma_F,\ \prec_F,\ \Omega_F,\ \tau_F}_{\sigma_F \oplus \{m\}, \prec_F \oplus A(m), \Omega_F \oplus \{m\} \ominus B(m), \{m\}}.$$

The semantic effect of the synchronous communication is a distributed assignment statement. Therefore,

$$T \equiv W^{\sigma_F,\ \prec_F,\ \Omega_F,\ \tau_F}_{\sigma_F \oplus \{m\}, \prec_F \oplus A(m), \Omega_F \oplus \{m\} \ominus B(m), \{m\}}.$$

Hence,

Backward Flush Send Axiom:

$$\{W^{\sigma_F,\ \prec_F,\ \Omega_F,\ \tau_F}_{\sigma_F\oplus\{\mathrm{m}\},\prec_F\oplus A(\mathrm{m}),\Omega_F\oplus\{\mathrm{m}\}\ominus B(\mathrm{m}),\{\mathrm{m}\}}\}\quad S_4 : \textbf{send} <\mathrm{BF}, \mathrm{data}> \textbf{on } F\ \{W\}.$$

Since the other three send axioms are similar in form, we omit the corresponding derivations.

Receive Axiom: A **receive** statement in an F-channel program,

$$\{\mathrm{P}\}\quad R_1\text{: } \textbf{receive} <\mathrm{mtype}, \mathrm{mdata}> \textbf{from } F\ \ \{\mathrm{Q}\},$$

is modeled by the following CSP communication statement:

$$\{\mathrm{P}\}\quad r_1\text{: } B?(<\mathrm{mtype}, \mathrm{mdata}>, \rho_F)\quad \{Q\}.$$

We can easily derive the Flush Channel Receive Axiom from the simulated CSP code. Since the **receive** allows a miracle, the postcondition Q is miraculous in isolation.

Satisfaction Rule: To derive the Flush Channel Satisfaction Rule, we begin with the Synchronous Satisfaction Rule for s_b and r_1.

$$P \land \mu \in (\sigma_F \ominus \rho_F) \land Pred(\mu) \subseteq \rho_F \Rightarrow Q^{<\mathrm{mtype},\mathrm{mdata}>,\ \rho_F}_{<\mu.type,\mu.data>,\ \rho_F\oplus\{\mu\}}.$$

Since P and Q are unspecified, the above implication may not be valid. It is necessary, therefore, to derive a satisfaction proof to ensure the axiomatic technique for F-channels is sound. First, we introduce $MTEXT$, a new variable, to replace μ. Second, the condition to transmit a message in B is based on the predecessor set of the message as defined in Chapter 1.2. Therefore,

$$Pred(\mu) \equiv Pred(MTEXT) \subseteq \rho_F \Rightarrow \forall m : m \in \sigma_F \land m \prec\!\!+_F MTEXT :: m \in \rho_F$$

Lastly, the **foreach** loop cannot continue indefinitely. Since messages are transmitted to B in a FIFO basis, some message must be available for delivery if $(\sigma_F \ominus \rho_F) \neq \emptyset$. The new satisfaction rule follows.

Flush Channel Satisfaction Rule:

For every F-channel receive

$$\{P\} \quad R_1 : \textbf{receive} <\text{mtype}, \text{mdata}> \textbf{from } F \; \{Q\},$$

verify the following to establish satisfaction:

$$P \;\wedge\; (MTEXT \in \sigma_F \ominus \rho_F) \;\wedge\; (\forall m : m \in \sigma_F \wedge m \prec\!\!+_F MTEXT :: m \in \rho_F)$$
$$\Rightarrow Q^{<\text{mtype},\text{mdata}>,\ \rho_F}_{MTEXT,\ \ \rho_F \oplus \{MTEXT\}}.$$

Non-interference Rule: We derive the Flush Channel Non-interference Rule from the non-interference proof of the CSP program. In B, consider all but the communication statements. None of these statements can interfere with parallel assertions in other processes as none of these statements update any variables. In addition, none of the assertions in B can interfere with parallel statements in other processes as the variables within the assertions are updated by synchronous communications involving B. The communication statements within B match communication statements in S and R. Therefore, non-interference proofs of S and R establish the non-interference proof of process B.

For every assertion A and for every statement S (not in B) that is a parallel assignment, simulation of a **send**, or simulation of a **receive**, we must prove the first step of the Synchronous Non-interference Rule. We, therefore, define

$$\{A \;\wedge \text{pre(S)}\} \quad \text{S} \quad \{A\}$$

as the first step in the Flush Channel Non-interference Rule.

For every assertion A and matching synchronous communication statements, we must prove the second step in the Synchronous Non-interference Rule. Again, for example, consider s_4 and r_b, and suppose that m represents <**BF**, data>. We must prove

$$A \ \wedge \ \text{pre}(s_4) \Rightarrow A^{\sigma_F, \quad \prec_F, \quad \Omega_F, \quad \tau_F}_{\sigma_F \oplus \{\text{m}\}, \prec_F \oplus A(\text{m}), \Omega_F \oplus \{\text{m}\} \ominus B(\text{m}), \{\text{m}\}}.$$

This implication is equivalent to proving

$$\{A \ \wedge \ \text{pre}(s_4)\} \quad s_4 : \textbf{send} \ \text{<}\textbf{BF}, \text{data>} \ \textbf{on} \ F \quad \{A\},$$

which is equivalent to validating the first step in the non-interference proof. We omit the details of the other three simulated **send** statements, s_1—s_3, as the outcome is the same.

Now consider the second step of the Synchronous Non-interference Rule for s_b and r_1.

$$A \ \wedge \text{pre}(r_1) \ \wedge \ \mu \in (\sigma_F \ominus \rho_F) \ \wedge \textit{Pred}(\mu) \subseteq \rho_F \Rightarrow A^{\text{<mtype,mdata>}, \ \rho_F}_{\text{<}\mu.type, \mu.data\text{>}, \ \rho_F \oplus \{\mu\}}.$$

As in the satisfaction case, we introduce *MTEXT*, a new variable, for μ and substitute $(\forall m : m \in \sigma_F \wedge m \prec\!\!+_F \textit{MTEXT} :: m \in \rho_F)$ for $\textit{Pred}(\textit{MTEXT}) \subseteq \rho_F$. Therefore,

$$A \wedge \text{pre}(r_1) \wedge (\textit{MTEXT} \in \sigma_F \ominus \rho_F) \wedge (\forall m : m \in \sigma_F \wedge m \prec\!\!+_F \textit{MTEXT} :: m \in \rho_F)$$
$$\Rightarrow A^{\text{<mtype,mdata>}, \rho_F}_{\textit{MTEXT}, \qquad \rho_F \oplus \{\textit{MTEXT}\}},$$

which is the second step in the Flush Channel Non-interference Rule.

All the rules and axioms in the F-channel axiomatic proof system have been derived from a CSP program that simulates an F-channel. We, therefore, conclude that the axiomatic proof methodology for processes communicating with F-channels is sound and relatively complete. ∎

3.10 Generalization of Conventional Asynchronous Communication

A benefit in using F-channels is that they can model reliable datagram and virtual circuit communication; F-channels are a true generalization of these conventional inter-process communication regimes. To model reliable datagrams, the sender only transmits ordinary messages. To model a virtual circuit, the sender can transmit only two-way flush messages. In the following two sections, we prove that the axiomatic proof system above is a true generalization of the verification process for communication with reliable datagrams and virtual circuits.

3.10.1 Relationship between Reliable Datagrams and Flush Channels

In this section, we prove the equivalent relationship between ordinary messages transmitted on an F-channel and reliable datagrams. Although the proof rules in Section 3.3.2 pertain to *Un*reliable Datagrams, we can use them for the reliable case as well.

Lemma 12 *In F-channel communication, $\prec_F$ is empty if the sender is restricted to transmitting only ordinary messages.*

Proof: We establish this result by induction on the number of ordinary messages transmitted in the system. The basis case, that only one ordinary message is transmitted, is trivially true.

Assume that n ordinary messages have been transmitted and $\prec_F = \emptyset$. Consider the transmission of m, the $(n+1)st$ ordinary message. By the Ordinary Message Send Axiom, at the transmission of m, $\prec_F$ is updated to include $A(m)$ where $A(m) = \{(x, m) \mid x \in \tau_F\}$. In a flush system restricted to ordinary message transmissions, $\tau_F = \emptyset$. Therefore, by the inductive hypothesis, $\prec_F = \emptyset$ after the transmission of m. ∎

Theorem 12 *Flush channels generalize reliable datagrams.*

Proof: The implicit variables which model those messages sent and received in the proof rules for both unreliable datagrams and F-channels are multisets. In addition, they both adhere to network axioms that insist $\rho_F \subseteq \sigma_F$, i.e. the Unreliable Datagram Network Axiom and the Flush Channel En Route Property. Flush channels also obey the Order Property. By Lemma 12, however, $\prec_F$ is empty. Hence, the Order Property places no restraints on the delivery order; if a message is available at the destination, it can be delivered.

The send axioms for unreliable datagrams and ordinary messages in F-channels include the assignment $\sigma_{D/F} := \sigma_{D/F} \oplus \{\text{msg}\}$. A transmission of an ordinary message also updates $\prec_F$ and Ω_F. As discussed, $\prec_F$ remains empty. The Ordinary Message Send Axiom updates Ω_F to include the newly transmitted message. This variable, however, is superfluous as Ω_F is never needed in an F-channel that only allows ordinary message transmissions. Therefore, the two send axioms for the different communication paradigms are equivalent. The Unreliable Datagram Receive Axiom and the Flush Channel Receive Axiom are trivially equivalent.

Consider the satisfaction rules of the two communication paradigms. The first, second, and implication clauses in the rules are equivalent. The third additional clause in the Flush Channel Satisfaction Rule refers to the delivery order. Since $\prec_F$ is empty, the clause has no bearing on the satisfaction proof.

In both communication paradigms, the proof of non-interference requires two steps. The first step is identical in the axiomatic proof methodology for unreliable datagrams and F-channels. Consider the second step in the non-interference rules of these two verification methodologies. The first, second, third, and implication clauses are trivially identical. As before, the fourth extra clause in the Flush Channel Non-interference (Receive) Rule is vacuously true. In conclusion, transmitting only ordinary messages on an F-channel simulates the transmission of messages on a reliable datagram. ∎

3.10.2 Relationship between Virtual Circuits and Flush Channels

The previous section showed that F-channels generalize reliable datagrams. The following discussion is similar. In this section, we prove that the axiomatic proof system for F-channels is a generalization of the verification process for virtual circuits.

Lemma 13 *In F-channel communication, if the sender is restricted to transmitting only two-way flush messages, then $\prec_F$ is a total order equivalent to the transmission order.*

Proof: We proceed by induction on the number of two-way flush messages transmitted across the F-channel. The basis case, that only one two-way flush message is transmitted, is vacuously true.

Assume that $\prec_F$ is a total order, identical to the transmission order, after n two-way flush transmissions. At the transmission of the nth two-way flush message, the Two-way Flush Send Axiom assigns the nth two-way flush to the free set. Consider the transmission of m, the $(n+1)st$ two-way flush message. By the send axiom, $\prec_F$ is augmented to include $C(m)$ where $C(m) = \{(x, m) \mid x \in \Omega_F\}$. Since Ω_F is a singleton, the nth two-way flush message, $\prec_F$ is updated with a single link that ensures the nth two-way flush message is delivered before m. By the inductive hypothesis, $\prec_F$ is a total order equivalent to the transmission order. ∎

Theorem 13 *Flush channels generalize virtual circuits.*

Proof: The implicit variables which model those messages sent and received in the proof rules for virtual circuits are sequences, guaranteeing a network axiom of $\rho_V \leq \sigma_V$. By Lemma 13, $\prec_F$ is a total order in a flush communication system that only transmits two-way flush messages. Thus, $\prec\!+_F$, the transitive closure of $\prec_F$, is a total order as well. Hence, the Virtual Circuit Network Axiom and the Flush Channel Order Property, when $\prec\!+_F$ is a total order, are equivalent; both communication paradigms will deliver messages in the transmission order.

The Virtual Circuit Send Axiom updates σ_V by appending the newly transmitted message. The Two-way Flush Send Axiom updates $\prec_F$ by, essentially, appending the newly transmitted message to the total order. Therefore, the send axioms of the two communication paradigms are equivalent. As in the datagram case, the receive axiom for these two asynchronous communication types are trivially equivalent.

Consider the satisfaction rules of the two communication paradigms. The first, second, and implication clauses in the rules are trivially equivalent. The third clause in the Virtual Circuit Satisfaction Rule states that $MTEXT$ must be the earliest message transmitted and not yet received. The third clause in the Flush Channel Satisfaction Rule, due to Lemma 13, ensures this property as well. Thus, the two satisfaction rules are equivalent.

As before, the first step in the non-interference rules of the two communication paradigms are identical. The first, second, third, and implication clauses are trivially identical. The fourth clause, of both rules, verifies the messages are delivered in a total order. Hence, transmitting only two-way flush messages on an F-channel simulates the transmission of messages on a virtual circuit. ∎

3.10.3 A Comment on the Generalization

It is important to realize that the generalization of reliable datagrams and virtual circuits to flush communication channels comes at a non-trivial cost in terms of our ability to reason about distributed systems. If virtual circuits are the communication mechanism in a distributed system, then the structure of the delivery partial order is fully known. This is independent of how a program chooses to pass data across a virtual circuit. The partial order is a total order. When we use the full generality of an F-channel, we build the structure of the partial order "on the fly." That structure cannot be known statically. Although this creates extra flexibility in the system, we feel that the complexity makes formal proof rules for flush systems imperative.

I don't want to achieve immortality through my work,
I want to achieve it through not dying.
Woody Allen

Chapter 4

Concluding Remarks

In this thesis, we investigated implementation and verification issues for flush communication channels. F-channels generalized the communication paradigms that enforced no delivery order (unreliable datagrams) and total delivery order (virtual circuits). In communication with an F-channel, the programmer defined the delivery order of each message in relation to other messages transmitted on the channel. Throughout the thesis, our formalization of the inherent partial order for message delivery facilitated our understanding of the dynamic, and possibly complex, delivery order.

From the system's perspective, an effective implementation that supported a dynamic delivery order specified during execution was not obvious. We reviewed two implementation protocols in the literature for F-channel communication and discussed their drawbacks. Understanding the partial order of messages intrinsic in F-channel message transmissions assisted us in developing the "WaitFor" technique. We presented the protocol and proved its correctness by validating that the protocol faithfully obeyed safety and liveness behavioral properties. The correctness of the WaitFor technique also served as the missing validity proof for the Three Counter technique, as we proved the functional equivalence of these two protocols.

In regards to flow control issues in message transmissions, we considered the constraints of finite buffer space and limited sequence numbers. In both matters, the partial delivery order precluded the use of conventional solutions. We presented solutions to bounding buffers and sequence numbers, acknowledged the flaws in the solutions, and argued that there is no preferable alternative. We then included bounding considerations in our WaitFor technique and proved the modified protocol was correct as well.

We presented results obtained from a simulation of F-channel message transmissions. As real-world examples naturally formed batches of ordinary messages and an associated flush message of a given type, our simulator considered message passing scenarios that partitioned ordinary messages into batches. After we reviewed the performance of virtual circuit communication, we plotted simulation results that considered three experimental parameters: degree of order, utilization, and number of links. All the results demonstrated that F-channel data transmission was faster than virtual circuit data transmission. Furthermore, the performance of forward flush and backward flush batching scenarios were quite similar, but substantially better than batching the ordinary messages with two-way flush messages. The two-way flush batching scenario, however, continued to outperform a virtual circuit. In conclusion, partially ordered message delivery allowed the possibility of higher bandwidth communication.

In order to validate the simulation results, we presented stochastic analysis of the three batching scenarios. We first reviewed the derivation of the expected resequencing delay of a message transmitted across a virtual circuit. The subsequent three sections considered the three batching scenarios in detail. In the forward flush batching case, the resequencing delay of an ordinary message was zero, while the mean resequencing delay of a forward flush message was identical to the mean resequencing delay of a message transmitted across a virtual circuit. It is to be expected, and shown to be correct, that the mean resequencing delay of a message in the forward flush batching scenario was a portion of the mean resequencing delay of a message transmitted across a virtual circuit. The analysis of the expected resequencing delay of a message in the two-way flush and backward flush batching scenarios was more complicated. In fact, approximations for the probabilities of distinct messages in transit were necessary in order to validate the simulation results.

The performance results showed that F-channels offered promise of ultra-high bandwidth communication over multiple physical paths. The programmer had the flexibility to choose the least amount of delivery order restrictions required to obtain the best performance in message transmissions. Programming with a system that communicated with an F-channel was, however, more complex than the conventional virtual circuit paradigms. To handle the additional complexity in the system, we developed an axiomatic verification methodology for F-channel communication.

We reviewed the conventional axiomatic proof methodology for synchronous communication with CSP and for asynchronous communication with unreliable datagrams and virtual circuits. We extended the methodology to F-channel communication by constructing the dynamic delivery order requirements within the axiomatic proof methodology. Though the addition of the delivery order construction increased the complexity of the verification methodology, we proved the axiomatic technique was sound and relatively complete. Lastly, we proved the equivalence of the axiomatic proof rules for F-channels and those for reliable datagrams and virtual circuits, demonstrating that F-channels could model these two conventional communication paradigms.

The use of flush communication channels provided a greater potential for concurrency in message passing than the use of virtual circuit communication, without the programming disadvantages of unreliable datagram communication. In F-channel communication, the programmer had the ability to simulate a virtual circuit or a reliable datagram; the programmer chose a partial order for the message delivery order that best fit the needs of the application. F-channels allowed the flexibility to relax the delivery order restrictions in virtual circuit communication and, hence, increased the rate of data transfer.

The results in this thesis suggested other possibilities of future work. From the implementation results, analytic error bounds for the approximate expected resequencing delays are needed. We are convinced that closed-form solutions for the two-way flush and backward flush batching scenarios cannot be obtained. For this reason, our approximation method is worthwhile. Error bounds on the approximation, however, would strengthen the results presented.

In the bounded WaitFor technique, we transmitted a *dummy* two-way flush message to reset the variables in the system when all messages had been ACKed and the transmission condition continued to be false. This *dummy* message was necessary to avoid deadlock. In effect, the transmission of the message synchronized the sender and receiver. In a system that thrives on concurrency, avoiding synchronizations was advantageous. We plan to consider other possibilities for a bounded WaitFor technique that do not require any synchronizations. Lastly, we want to implement a prototype for F-channel communication.

In verifying F-channel applications, we want to consider a second proof methodology. The axiomatic operational proof methodology in this thesis relied on nonlocal reasoning for correctness. Whether an application verified its intentions or not depended on global arguments built after the processes were annotated. We should not construct our view of a distributed system by adding order, as we must do with any definition of global state. Instead, we should reason with events and states in a distributed system using causal order as defined by Lamport's "happened before" relation [Lam78]. In a causal proof methodology, we do not need to consider the communication state. Instead, we look into the possible causal relationships between senders and receivers.

The motivation for a causal proof methodology for F-channel communication is to simplify testing [Llo91]. Due to a dynamic, and possibly complex, delivery order in F-channel communication, verifying an application and testing the correctness of the assertions will further build our confidence that the application satisfies our expectations. Our initial consideration of a causal proof methodology for F-channel communication developed a causal reasoning technique that is correct for a two process system. Generalizing the technique to any number of processes produced problems. We believe that a causal proof methodology for F-channel communication must abandon the definition of no auxiliary variables in the reasoning process. We plan to consider this area further in the future.

Bibliography

[AFR80] K.R. Apt, N. Francez, and W.P. De Roever. A proof system for communicating sequential processes. *ACM Transactions on Programming Languages and Systems*, 2:359–385, 1980.

[Ahu90] M. Ahuja. Flush primitives for asynchronous distributed systems. *Information Processing Letters*, 34:5–12, 1990.

[Ahu91] M. Ahuja. An implementation of F-channels, a preferable alternative to FIFO channels. In *Proceedings of the 11th International Conference on Distributed Computing Systems*, pages 180–189, 1991.

[AR87] S. Agrawal and R. Ramaswamy. Analysis of the resequencing delay for $M/M/m$ systems. In *Proceedings of the 1987 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 27–35, 1987.

[AS85] B. Alpern and F.B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, 1985.

[AVS91] M. Ahuja, K. Varadhan, and A.B. Sinha. Flush message passing in communicating sequential processes. In N. Rishe, S. Navathe, and D. Tal, editors, *Parallel Architectures*, pages 31–47. IEEE Computer Society Press, 1991.

[Cho89] S. Chowdhury. The mean resequencing delay for $M/H_k/\infty$ systems. *IEEE Transactions on Software Engineering*, 15(12):1633–1638, 1989.

[CK91] T. Camp and P. Kearns. Proof rules for flush channels: An axiomatic approach. Technical Report WM–91-2, The College of William and Mary, 1991.

[CKA93] T. Camp, P. Kearns, and M. Ahuja. Proof rules for flush channels. *IEEE Transactions on Software Engineering*, 19(4):366–378, April 1993.

[CL85] K.M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, 1985.

[Coo78] S.A. Cook. Soundness and completeness of an axiom system for program verification. *SIAM Journal of Computing*, 7(1):70–90, 1978.

[Dij76] E.W. Dijkstra. *A Discipline of Programming*. Prentice Hall, Englewood Cliffs, 1976.

[Hoa69] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, 1969.

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

[KC91] P. Kearns and T. Camp. An implementation of flush channels based on a verification methodology. Technical Report WM–91–6, The College of William and Mary, 1991.

[KCA92] P. Kearns, T. Camp, and M. Ahuja. An implementation of flush channels based on a verification methodology. *Proceedings of the 12th International Conference on Distributed Computing Systems*, pages 336–343, 1992.

[Kle75] L. Kleinrock. *Queuing Systems*, volume 1. John Wiley & Sons, Inc., 1975.

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

[Lam88] L. Lamport. Control predicates are better than dummy variables for reasoning about program control. *ACM Transactions on Programming Languages and Systems*, 10(2):267–281, 1988.

[LG81] G.M. Levin and D. Gries. A proof technique for communicating sequential processes. *Acta Informatica*, 15:281–302, 1981.

[LK91] W.S. Lloyd and P. Kearns. Using tracing to direct our reasoning about distributed programs. In *Proceedings of the 11th International Conference on Distributed Computing Systems*, pages 552–559, 1991.

[Llo91] W.S. Lloyd. *Causal Reasoning about Distributed Programs.* PhD thesis, The College of William and Mary, 1991.

[LS84] L. Lamport and F.B. Schneider. The 'Hoare logic' of CSP, and all that. *ACM Transactions on Programming Languages and Systems*, 6(2):281–296, 1984.

[MC81] J. Misra and K.M. Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, SE-7:417–426, 1981.

[OG76] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6:319–340, 1976.

[OL82] S. Owicki and L. Lamport. Proving liveness properties of concurrent programs. *ACM Transactions on Programming Languages and Systems*, 4(3):455–495, 1982.

[Ros73] S.M. Ross. *Introduction to Probability Models.* Academic Press, Inc., 1973.

[Sou84] N. Soundararajan. Axiomatic semantics of communicating sequential processes. *ACM Transactions on Programming Languages and Systems*, 6(4):647–662, 1984.

[SS84] R.D. Schlichting and F.B. Schneider. Using message passing for distributed programming: Proof rules and disciplines. *ACM Transactions on Programming Languages and Systems*, 6(3):402–431, 1984.

[Tan89] A.S. Tanenbaum. *Computer Networks*. Prentice-Hall, Inc., 1989.

[YN86] T-S. Yum and T-Y. Ngai. Resequencing of messages in communication networks. *IEEE Transactions on Communications*, COM–34(2):143–149, 1986.

VITA

Tracy Camp was born in Detroit on September 27, 1964. After graduating from Lakeland High School in Milford, Michigan, she attended Kalamazoo College in her home state and received a B.S. degree in Mathematics, June 1987. She then continued her education at Michigan State University in Lansing and received an M.S. degree in Computer Science, March 1989. In September 1989, Ms. Camp entered the College of William and Mary and expects to receive her doctorate in Computer Science in 1993. She will join the faculty at the University of Alabama at Tuscaloosa in September 1993.