

# Hierarchical Channels and the Relative Speed of Messages <sup>1</sup>

Ravi Prakash  
Department of Computer Science  
University of Rochester  
Rochester, NY 14627-0226.

Mohan Ahuja<sup>2</sup>  
115 N. Doheny Drive, # 307  
Los Angeles, CA 90048-2829.

<sup>1</sup>This paper is an enhanced version of the paper [4] presented at SPDP, 1992.

<sup>2</sup>This research was supported in part by the National Science Foundation under grants MIP-9111045 and MIP-9296204.

## Abstract

FIFO communication is expensive to support. It is also quite restrictive for several applications. In this paper, we study the concept of *relative speed of messages* and a hierarchy of such speeds which results in a *hierarchical channel*. In *hierarchical channels* messages sent at lower levels of the hierarchy cannot be overtaken by message sent at a higher level. But lower level messages can overtake higher level messages. The flexibility provided by *hierarchical channels* can be exploited by several distributed applications. We present examples to illustrate how FIFO ordering constraints can be relaxed for such applications using *hierarchical channels*, leading to increased concurrency. We present a new algorithm to implement *hierarchical channels*. The algorithm employs selective flooding, and is tolerant to link-failures. It is possible to calculate the statistical distribution of message propagation delays for a selective flooding implementation of *hierarchical channels*, as shown in the paper. So, they can be employed to implement statistical channels. If *hierarchical channels* are used for communication, assertions can be made about end-to-end message ordering characteristics. Thus, *hierarchical channels* subsume message overtaking characteristics of point-to-point links, as well as send and receive queues at intermediate nodes. Hence, *hierarchical channels* can support stronger assertions about message ordering, yet provide greater flexibility than priority queues. Hierarchical channels can also be used to implement priority queues. Compared to *F-channels*, *hierarchical channels* can model a greater variety of message orderings: all the way from FIFO message delivery to totally unordered message delivery.

**Keywords:** message ordering, FIFO communication, F-channels, hierarchical channels, selective flooding.

### Contact Author:

Ravi Prakash  
Department of Computer Science  
University of Rochester  
Rochester, New York 14627-0226, U.S.A.  
Phone: (716) 275-5492  
Fax: (716) 461-2018  
e-mail: [prakash@cs.rochester.edu](mailto:prakash@cs.rochester.edu)

# 1 Introduction

Communication between processes in a distributed computation has been usually modeled as being reliable with unpredictable, yet finite, message transmission time. Such a communication model is relevant for database applications that require ordered and reliable flow of data. The ordering constraint is usually equated with *FIFO* message delivery.

However, many applications need communication with relaxed reliability and/or ordering constraints. For example, multimedia communication over computer networks requires handling audio and video streams of data that are played back in real-time at the destination site. These streams can tolerate some loss of information as long as these losses do not cause a degradation in the desired *quality of service* (QoS) for the application [9, 25]. Also, in some instances, messages need not necessarily be delivered in *FIFO* order. Therefore, the ordering and reliability constraints imposed by database applications can be relaxed for multimedia applications. So, communication paradigms suitable for database applications may not necessarily be suitable for other applications. New communication paradigms need to be defined these applications.

In [1], Ahuja proposed *F-channels* for asynchronous distributed systems that support a set of communication primitives. The primitives are *o-send*, *t-send*, *f-send* and *b-send* for sending four types of messages to be referred to as *o* (ordinary message), *t* (*two-way-flush*), *f* (*forward-flush*) and *b* (*backward-flush*) respectively. A message that is either a *t*, *f*, or *b* message is referred to as a *Flush* message. An *o* message can overtake other messages and other messages can overtake it, if not forbidden by *Flush* messages. A *t* message does not overtake any message and no message overtakes it. An *f* message does not overtake any message. No message overtakes a *b* message. For a given sequence of messages sent by process *p* to process *q*, an *F-channel*  $c_{p,q}$  does not permit *all* the possible sequences of message reception in the *FIFO* to non-*FIFO* range. However, for many important classes of problems, *e.g.*, echo algorithms, snapshot algorithms, these primitives do away with most of the unwanted restrictions imposed by the *FIFO-channels* and permit a higher degree of concurrency. At the same time, properties of solutions to these problems using *F-channels* can be proved as easily as for those using *FIFO-channels*.

In this paper we study the concept of *relative speed of messages* and a hierarchy of such speeds resulting in *hierarchical channels*. Hierarchical channels permit finer granularity in specifying con-

currency in message passing compared to *F-channels*. Hierarchical channels can provide application developers with the ability to specify an entire spectrum of message ordering constraints from *FIFO* to completely unordered message delivery. Such channels can also be employed to implement statistical channels with desired quality of service characteristics, and support expedited data transfer.

Hierarchical channels were first introduced in [2]. An implementation of *hierarchical channels*, using counters, was presented in [18, 19]. The novel contributions of this paper are in three areas: (i) We present a new algorithm for implementing hierarchical channels, and prove the correctness of the algorithm. (ii) We present examples to show how communication primitives based on hierarchical channels can lead to increased concurrency and flexibility in distributed applications. The performance improvement and increased flexibility, as illustrated by the examples, are independent of the algorithm used to implement hierarchical channels. (iii) We model the performance of the proposed algorithm. Based on the analytical model we argue that hierarchical channels can find application in designing statistical channels, and for achieving expedited data transfer.

This paper presents an algorithm to implement hierarchical channels using *selective flooding*. The previous counter based implementation [19] incurs an increase in message delay at the receiver's node, and requires each message to carry dependency information about all messages sent prior to it at lower levels. The selective flooding algorithm requires extra messages. It is suitable for environments in which an increase in the number of messages in the underlying network is more acceptable than an increase in message delay associated with the counter based implementation. In addition to avoiding delays at the receiver node, the proposed selective flooding algorithm has the shortest possible communication delay for a message (for a given topology and message traffic).

In Section 2 we define the concepts of *relative speed of messages* and a hierarchical channel [2]. Earlier papers on *hierarchical channels* [2, 18, 19] have focused primarily on defining the semantics of hierarchical channels. We go a step further by illustrating how strong, yet flexible, communication primitives provided by hierarchical channels can be used by distributed applications for increased concurrency. In Section 3 we present examples of the use of hierarchical channels in designing distributed applications. The examples demonstrate the versatility of hierarchical channels. The increase in concurrency of distributed applications can be achieved without significantly complicating the provability of their correctness. Section 4 describes the system topology and

presents the algorithm to implement hierarchical channels using selective flooding. The correctness of the algorithm is proved and performance of the hierarchical channels is analyzed. The proposed algorithm is shown to be resilient to link-failures. Section 5 describes how the selective flooding algorithm for hierarchical channels can be used to implement statistical channels. The ability to probabilistically predict the performance characteristics of hierarchical channels helps in such implementations. The earlier counter based approach to hierarchical channels [18, 19] cannot be used to implement statistical channels. In Section 6 we compare hierarchical channels with *F-channels*, expedited data transfer protocols and priority queues. The conclusion is presented in Section 7. In the Appendix we show how the implementation of hierarchical channels (presented in Section 4.5) needs to be modified only slightly to implement priority queues and expedited data transfer protocols.

## 2 Definitions

We assume a system consisting of a set of processes and communication channels between them. Information is exchanged between processes through messages. The sender of a message specifies the destination. The destination process accepts messages if it is ready to receive and if the message semantics permit such reception. We assume that no messages are lost, corrupted, or duplicated in transit. They are delivered in a finite time during normal operation.

### 2.1 Message Speed

In systems with a global clock a message's speed is inversely proportional to the time taken by the message to travel from the sender process to the receiver process. Such a definition of message speed cannot be used for an asynchronous distributed system (henceforth referred to as a distributed system) due to the absence of a global clock.

### Relative Speed of Messages

*Relative speed of communication*, henceforth to be referred to as *relative speed*, can be defined based on the concept of *overtaking*.

In [1, 2], message overtaking is defined as follows:

**Definition 1 (Overtaking)** Let  $m_i$  be a message sent by process  $p$  to process  $q$ . Message  $m_j$  is said to overtake  $m_i$  (both sent along the same channel) if  $m_i$  is sent before  $m_j$ , and  $m_j$  is received before  $m_i$ .

**Definition 2 (Relative Speed)** Relative speed of a message  $m'$ , sent by process  $p$  to process  $q$ , is defined to be at least as fast as that of  $m''$ , also sent by  $p$  to  $q$ , if  $m''$  cannot overtake  $m'$ .

## 2.2 Hierarchy of Relative speeds of Messages and the Resulting Hierarchical Channel

We assume that a process may send messages to another process along a number of unidirectional channels rather than just one channel. A channel represents a directed path between two processes. The relative speeds of channels can be characterized by those of the messages traveling along them. We use  $c'_{p,q}$ ,  $c''_{p,q}$ , etc. to represent individual channels from  $p$  to  $q$ , and  $c_{p,q}$  to collectively represent all the channels from  $p$  to  $q$ . Channels  $c'_{p,q}$ ,  $c''_{p,q}$  etc. will be referred to as *c\_channels* (component channels.) When the source and destination processes are obvious from the context, the subscripts will be dropped. Given two *c\_channels*,  $c'$  and  $c''$ , from  $p$  to  $q$ , the only possible assertion we can make about them, in the absence of a global clock, is either “ $c'$  is at least as fast as  $c''$ ” or “ $c'$  is at most as fast as  $c''$ ” using the following definition.

**Definition 3 ( $c'$  is at least as fast as  $c''$ )**  $c'$  is at least as fast as  $c''$  (or  $c''$  is at most as fast as  $c'$ ) when every message  $m'$  sent along  $c'$  is at least as fast as (and so cannot be overtaken by) any message  $m''$  sent along  $c''$ , i.e., if  $m'$  is sent along  $c'$  before  $m''$  is sent along  $c''$  then  $m'$  is received before  $m''$ .

Note that if  $c'$  is at least as fast as  $c''$  then messages sent along  $c'$  may or may not overtake those sent along  $c''$ .

**Definition 4 (General Hierarchical Channel)**  $c_{p,q}$  is said to be a general hierarchical channel if each of its *c\_channels* can be assigned a level such that a *c\_channel* at level  $u$ , represented as  $c^u_{p,q}$  is at least as fast as a *c\_channel* at level  $v$ , represented as  $c^v_{p,q}$ , where  $u \leq v$ .

In the above definition a *c\_channel* need *not* necessarily be *FIFO*. When the rules for overtaking within  $c^v_{p,q}$  are the same for all  $v$ ,  $c_{p,q}$  is said to be a *uniform hierarchical channel*, otherwise a *non-uniform hierarchical channel*. Since our goal in this paper is to study the hierarchy of message

speeds we will assume that channels are *uniform hierarchical channels* and that each  $c_{p,q}^v$  is *FIFO*. We will refer to it as a hierarchical channel.

**Definition 5 (Hierarchical Channel)**  $c_{p,q}$  is a hierarchical channel if it is a general hierarchical channel and each of its *c\_channels* is *FIFO*.

Thus  $p$  sending a message to  $q$  along such a  $c_{p,q}$ , is equivalent to  $p$  sending the message over any one of the number, say  $n$ , of *c\_channels* such that each *c\_channel*  $c_{p,q}^u$  is at least as fast as  $c_{p,q}^v$ , where  $u \leq v$  and  $u, v = 0$  to  $n - 1$ .

If a hierarchical channel  $c_{p,q}$  is drawn as  $q \leftarrow p$ , implying that messages move from right to left, then it can be visualized as a multi-lane road in which a lane corresponds to a *FIFO c\_channel*  $c_{p,q}^v$  such that each lane is at least as fast as lanes to its right. Figure 1 illustrates this definition. Note that this definition permits a continuum of levels, i.e., level numbers are not restricted to be integers (but can be mapped to integers). Hence, given two levels  $u$  and  $v$  where  $u < v$ , there can be infinitely many levels between them. This provides an ability to program a wide range of message ordering from *FIFO* to non-*FIFO*; and poses problems in implementation which will be discussed later.

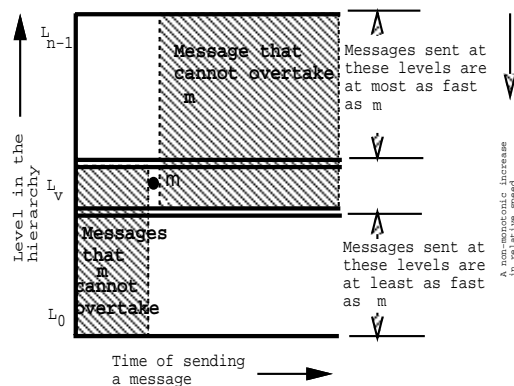


Figure 1: A hierarchical channel.

A message  $m$  is sent by executing  $send(level, destination, m)$  and is received without violating the semantics as specified by the level property defined next.

**Definition 6 (Level Property)** A message sent by  $p$  to  $q$ , along  $c_{p,q}$ , at level  $v$  cannot overtake messages sent at level  $u$  where  $u \leq v$  and may overtake messages sent at levels  $w$  where  $v < w$ .

### 3 Applications of Hierarchical Channels

We present examples of applications that bring forth the strengths and usefulness of hierarchical channels. These examples are independent of the method used to implement hierarchical channels, whether counter based [18, 19], or selective flooding based as proposed in Section 4. The first two examples demonstrate the flexibility provided by the *hierarchical channels*, when used with the *alternative command* of CSP [8]. The third example describes how hierarchical channels can be used to collect a consistent snapshot of a distributed application.

Later, in Section 5, we present an application that is implementation dependent. We describe how the proposed selective flooding based implementation of hierarchical channels can be exploited to implement statistical channels. The implementation of statistical channels using selective flooding is motivated by our performance analysis (Section 4.7) showing that lower the level of a message in the hierarchy, smaller the expected message propagation time.

#### 3.1 A Message Passing System

In this example three messages are sent by a source node to a destination. The level of a message is specified using the *level* variable and the destination using *dest*. The  $i^{\text{th}}$  message sent by the source will be referred to as  $m_i$ .

```
 $m_1.$     send(level+1, dest, message);  
 $m_2.$     [send(level, dest, message);  
        □  
        send(level+1, dest, message);  
        ]  
 $m_3.$     send(level+0.5, dest, message);
```

If the second alternative is chosen for  $m_2$ , then  $m_2$  is delivered after  $m_1$  because message transmission at a level is *FIFO*. Message  $m_3$ , being sent at a lower level than the preceding two, can overtake both or just the second of them. If the first alternative is selected for  $m_2$  then it is sent at a lower level than  $m_1$  and can overtake it. But  $m_3$  cannot overtake  $m_2$  though it can overtake  $m_1$ . So the possible orders in which the three messages can be received are:  $[m_1, m_2, m_3]$ ,  $[m_1, m_3, m_2]$ ,  $[m_3, m_1, m_2]$ ,  $[m_2, m_1, m_3]$ , and  $[m_2, m_3, m_1]$ . This multiplicity of the order of message receptions provides great flexibility in the design of message-passing systems and compares

favorably with the only possibility,  $[m_1, m_2, m_3]$ , available with *FIFO-channels*. Note that we have used real numbers to denote the levels.

### 3.2 Evaluation of Commutative Expressions

Consider the expression  $(a + b) * (c + d)$  which has commutative operations. It can be evaluated in any of the following ways:

- |                        |                        |
|------------------------|------------------------|
| 1. $(a + b) * (c + d)$ | 2. $(a + b) * (d + c)$ |
| 3. $(b + a) * (c + d)$ | 4. $(b + a) * (d + c)$ |
| 5. $(c + d) * (a + b)$ | 6. $(c + d) * (b + a)$ |
| 7. $(d + c) * (a + b)$ | 8. $(d + c) * (b + a)$ |

Let us consider a distributed memory system in which this expression has to be evaluated by node  $p$  while the values  $a, b, c$  and  $d$  are resident at node  $q$  and data communication between nodes is through messages. Once the instruction corresponding to this expression has been fetched and decoded by  $p$  it sends all the data-fetch requests to  $q$ . Subsequently, for  $0 < \alpha, 0 < \beta, 0 < \delta < \beta$ , the code to be executed by  $p$  and  $q$  can be written as follows:

$q::$	$[$	$send(l, p, a);$	$p::$	$[$	$recv(x);$	$recv(y);$	$x := x + y;$
		$send(l - \alpha, p, b);$			$recv(y);$	$recv(z);$	$y := y + z;$
		$send(l + \beta, p, c);$			$x := x * y;$		
		$send(l + \delta, p, d);$			$output(x);$		
		$]$			$]$		

Note that the first parameter of *send* is the level at which the message is sent. The code permits the expression to be evaluated as either one of the four alternatives 1, 2, 3, 4. Data need not necessarily be received in *FIFO* order for the expression to be evaluated correctly.

Hence, the commutativity and associativity of arithmetic operations can be exploited with hierarchical channels. *Operations can be programmed so as to trigger as soon as all their operands are available, regardless of the order in which they are obtained.*

### 3.3 A Distributed Snapshot Algorithm

Predicate evaluation is an important issue in distributed computation and in order to do so a consistent view of the system is required. Several algorithms [1, 5, 7, 12, 13, 16, 20, 23, 24] have

been proposed to determine consistent global snapshots of a system. In [5] Chandy and Lamport present one of the earliest and most elegant algorithms for a *FIFO* system. For non-*FIFO* systems, the algorithms presented in [7, 13, 24] are either inhibitory in nature [7] or have problems associated with log keeping [13, 24].

Using messages with specifiable levels, a non-inhibitory algorithm can be constructed for non-*FIFO* communication systems. The algorithm is a simple variation of [5]. Let us assume that the algorithm is assigned a level,  $l$ , such that all the application messages are sent at  $l$  or levels above  $l$ , while all the markers for checkpointing are sent at level  $l$ . As a result, no application message overtakes any marker, but an application message or a marker may overtake another application message. As in [5], the snapshot initiator takes its local snapshot and sends markers to all its neighbors before sending any other message. On receiving the first marker, all the other processes take their local snapshots and send markers to all their neighbors (except the ones that sent them the first marker). If there are  $n$  processes in the system, four integer vectors: *In*, *Out*, *Deficit* and *Marker*, each having  $n$  components are maintained by each process. All of them are initialized to zero and are used to determine the number of messages in transit in the snapshot,

When process  $p$  sends a message (application or marker) to process  $q$ , it increments  $Out[q]$  by one and tags this number with the message. When  $q$  receives a message from  $p$  it increments  $In[p]$  by one. If the message happens to be a marker the tag on the received message is stored in  $Marker[p]$ .  $Marker[p] - In[p]$  is equal to the number of messages in transit from  $p$  to  $q$  and is stored in  $Deficit[p]$  by  $q$ . Subsequently, every message received by  $q$  from  $p$  with a tag less than  $Marker[p]$  is a message in transit for the snapshot. Whenever such a message is received  $Deficit[p]$  is decremented by one. When  $Deficit[p]$  reduces to zero all messages in transit along the channel  $c_{p,q}$ , that form the state of  $c_{p,q}$  for the snapshot, have been received by  $q$ . The delay involved in the collection of channel states in our algorithm is no more than that involved when the *vector counter principle* suggested by Mattern [14, 15] is employed to detect the number of messages in transit using at most two control rounds. Our algorithm does not require such control rounds.<sup>1</sup> Besides, it is almost as simple as Chandy and Lamport's algorithm even though the communication-network is non-*FIFO*.

---

<sup>1</sup>The delay can be avoided if each process maintains a log of messages sent along every channel since the last snapshot and sends this log with the marker. But this would place high memory overheads on the algorithm.

## 4 Algorithm to Implement Hierarchical Channels

An implementation of *hierarchical channels* using counters has been presented in [18, 19]. The counter based implementation involves message buffering, and therefore, an increase in message latency at the receiver node until message reception will not violate the level property. Here we present an alternative algorithm to implement hierarchical channels using selective flooding. The proposed algorithm requires extra messages. It is suitable for environments in which an increase in the number of messages in the underlying network is more affordable than an increase in message delay associated with the counter based implementation, *e.g.*, real-time multimedia communication. In addition to avoiding delays at the receiver, the proposed algorithm has the shortest possible communication delay for a message (for a given topology and message traffic).

### 4.1 Bounding Number of Levels

Even though there can be infinitely many levels of relative message speed between a pair of processes, in reality they will need to be implemented as a finite number of levels. This is because for a counter based implementation only a finite number of counters can be carried with a message; and for a selective flooding based implementation as presented here, there are only a finite number of physical paths between any two nodes. An application program using hierarchical channels may still be presented with the illusion of having infinitely many levels at its disposal by employing a many-to-one mapping between the levels specified in the application program and the levels available.

### 4.2 Illustrative Example

We will describe the idea behind the selective flooding implementation of hierarchical channels with the help of Figure 2.

The hierarchical channel  $c_{p,r}$  can be decomposed into two contiguous hierarchical channels  $c_{p,q}$  and  $c_{q,r}$ . Hierarchical channel  $c_{p,q}$  consists of five paths referred to as paths 1 – 5. Similarly, hierarchical channel  $c_{q,r}$  is composed of four paths referred to as paths 6 – 9. Let process  $p$  send messages  $m_1$ ,  $m_2$ , and  $m_3$  to process  $r$  along channel  $c_{p,r}$ . The corresponding levels of the three messages are  $u$ ,  $v$ , and  $w$  such that  $u < v < w$ .

Messages sent at level  $u$  on  $c_{p,r}$ , are first sent to process  $q$  along paths 1 – 5, and then forwarded

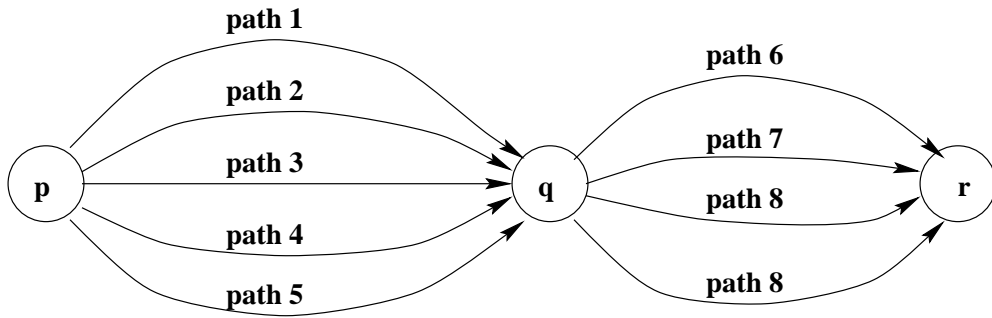


Figure 2: An example to illustrate selective flooding implementation of hierarchical channels.

from  $q$  to  $r$  along paths 6 – 9. Messages sent at level  $v$  on  $c_{p,r}$ , are first sent to process  $q$  along paths 1 – 3, and then forwarded from  $q$  to  $r$  along paths 6 – 8. Messages sent at level  $w$  on  $c_{p,r}$ , are first sent to process  $q$  along paths 1 and 2, and then forwarded from  $q$  to  $r$  along paths 6 and 7.

Let the order of sending the messages be  $m_2$  followed by  $m_1$ , and finally  $m_3$ . So,  $p$  sends one copy of  $m_2$  along each of the paths 1 – 3. The moment  $q$  receives the first copy of  $m_2$  it forwards one copy of  $m_2$  along each of the paths 6 – 8. All subsequent copies of  $m_2$  arriving at  $q$  are discarded. Message  $m_2$  is said to be delivered to process  $r$  the moment its first copy arrives at  $r$ . Subsequent copies of  $m_2$  received by  $r$  are discarded.

After sending  $m_2$ , process  $p$  sends  $m_1$  by sending one copy of  $m_1$  along each of the paths 1 – 5. Even though  $m_1$  is sent after  $m_2$ , copies of  $m_1$  sent along paths 4 and 5 may reach process  $q$  before copies of  $m_2$  reach there along paths 1 – 3. So,  $m_1$  can overtake  $m_2$  in the hierarchical channel  $c_{p,q}$ . As a result,  $q$  will send copies of  $m_1$  to process  $r$  along paths 6 – 9 before it sends copies of  $m_2$  to  $r$  along paths 6 – 8. As all the paths are *FIFO*,  $m_2$  cannot overtake  $m_1$  between  $q$  and  $r$ . Also, if  $m_1$  is received by process  $q$  after  $m_2$ ,  $m_1$  can overtake  $m_2$  in the hierarchical channel  $c_{q,r}$ .

When  $m_3$  is sent by  $p$  along paths 1 and 2, all copies of  $m_3$  will reach  $q$  only after all copies of  $m_1$  and  $m_2$  have reached  $q$ . Similar argument hold for  $m_3$  with respect to  $m_1$  and  $m_2$  along hierarchical channel  $c_{q,r}$ . So,  $m_3$  cannot overtake  $m_1$  and  $m_2$  along channel  $c_{p,r}$ .

Therefore,  $m_1$ , a message sent at a lower level, can overtake  $m_2$  which is sent at a higher level. However,  $m_3$ , which is sent at a higher level than both  $m_1$  and  $m_2$  cannot overtake either of them.

### 4.3 System Topology

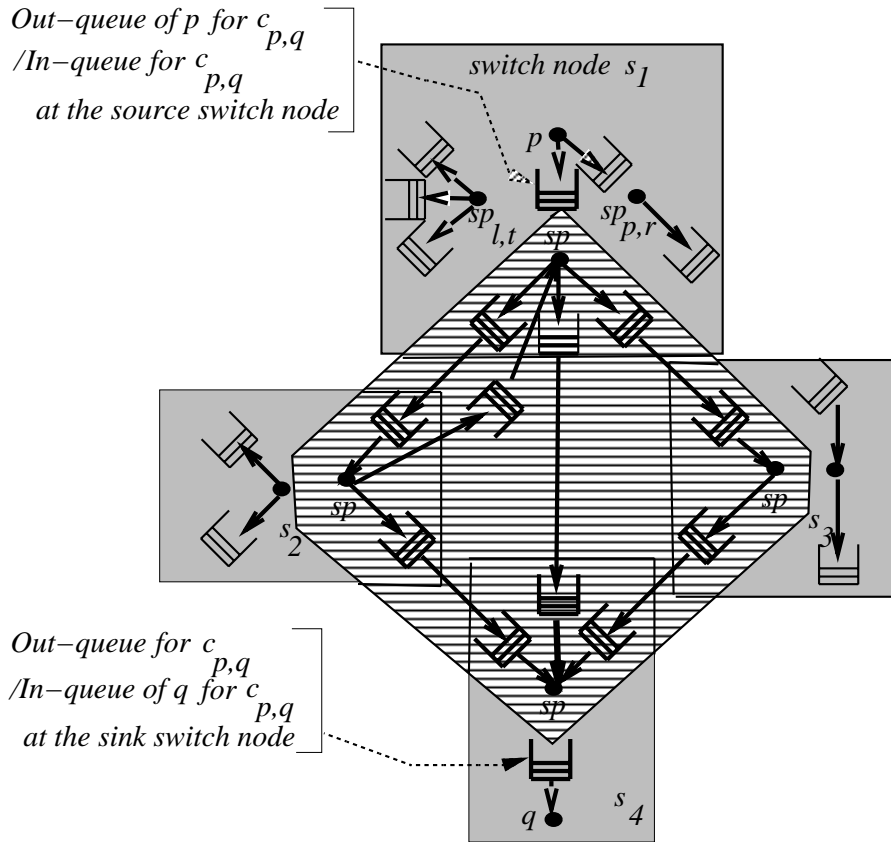
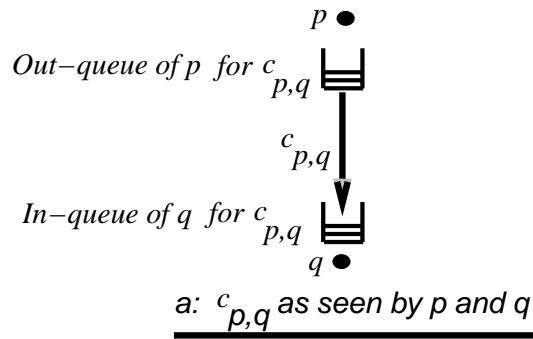
An example of the assumed system topology is given in Figure 3. We use the terminology first presented in [3]. The *u-net* refers to the underlying network consisting of all the paths between every pair of processes. Each path in the *u-net* is composed of zero or more channels, referred to as *u\_channels*, connecting the switch nodes on the paths. The subset of *u-net* connecting  $p$  to  $q$  on which messages sent along  $c_{p,q}$  travel is finite and directed and is called the *s-net* (switch network) for  $c_{p,q}$ . Channels in the *s-net* are referred to as *s\_channels*. In addition to paths for  $c_{p,q}$ , the *u-net* may also have other paths  $c_{p,r}$  and  $c_{q,p}$ , etc. If a *u\_channel* belongs to more than one *s-net*, for simplicity of expression we consider it as a separate *s\_channel* in each of the *s-nets* to which it belongs. We assume that multiplexing and demultiplexing at the sending and receiving ends, respectively, of *s\_channels* is done correctly.

Every switch node has a switch process  $sp$  for each channel  $c_{p,q}$  on whose *s-net* the node lies. The switch process  $sp$  handles messages sent at all possible levels,  $u$ , along  $c_{p,q}$ . Every in-coming *s\_channel* at  $sp$  ends in an *in\_queue* in which the received messages are queued. Every out-going *s\_channel* at  $sp$  has an *out\_queue* at its head. Messages to be sent out along an *s\_channel* are placed in the *out\_queue* from where they are forwarded along the channel in *FIFO* order. The switch process  $sp$  forwards a message on its way to its destination by transferring it from the appropriate *in\_queues* to the corresponding *out\_queues* for that level. In the context of  $c_{p,q}$ , the node on which process  $p$  executes will be referred to as the source node. Process  $p$  has an *out\_queue* into which it puts messages to be sent along  $c_{p,q}$ . The *out\_queue* of  $p$  is the only *in\_queue* in the *s-net* for  $c_{p,q}$ . This *out\_queue* is also one of the *in\_queues* (the *in\_queues* may be more than one due to possible cyclicity of the *s-net*) of  $sp$  at the source switch node. The node on which process  $q$  executes will be referred to as the sink node. Process  $q$  has an *in\_queue* from which it receives messages sent along  $c_{p,q}$ . The *s-net* for  $c_{p,q}$  has only one *out\_queue*, which is also one of the *out\_queues* (the *out\_queues* may be more than one due to possible cyclicity of the *s-net*) of  $sp$ , at the sink switch node. All these queues and *s\_channels* are *FIFO* in nature.

The assumption that all the *s\_channels* and queues are *FIFO* is meant for simplicity of expression. This assumption of *FIFO s\_channels* and queues can be relaxed so that channels can either be *FIFO* or preserve the level property<sup>2</sup> and queues can either be *FIFO* or be relaxed to satisfy

---

<sup>2</sup>Assumption of existence of an *s\_channel* which preserves level property may seem circular, but it is not since



: A switch node.

$sp, sp_{p,r}$  : Switch processes for  $c_{p,q}$  and  $c_{p,r}$ , respectively.

:  $s$ -net, a directed and finite network of channels that are either FIFO or satisfy level-property and that transfers messages for  $c_{p,q}$ . It has a path only if such path exists in the underlying network.

b:  $s$ -net and implementation of  $c_{p,q}$

Figure 3:  $c_{p,q}$  as seen by  $p$  and  $q$  and an example of the  $s$ -net of  $c_{p,q}$  and switch processes implementing

$c_{p,q}$ .

the level property. After this relaxation, the implementation and its proof of correctness remains the same. The relaxation of the *FIFO* property of *in\_queues* is required to support expedited data transfer, as described in Subsection 6.2.

#### 4.4 Basic Idea

Let there be  $n$  paths in the *s-net* for  $c_{p,q}$ . Let us assume  $u \leq v < w$ . The idea is that one copy of a message sent along  $c_{p,q}^v$ , for  $0 \leq v \leq (n - 1)$ , travels along each path in a subset  $\mathcal{C}^v$  of the paths in the *s-net*. *In\_queues* in which *sp* at a switch node receives a message sent along  $c_{p,q}^v$  will be referred to as  $C\_in\_v$ . *Out\_queues* into which *sp* at a switch node puts a message to be sent along  $c_{p,q}^v$  will be referred to as  $C\_out\_v$ . For  $u \leq v$ ,  $\mathcal{C}^v \subseteq \mathcal{C}^u$ . Hence,  $C\_in\_v$  is a subset of  $C\_in\_u$  and  $C\_out\_v$  is a subset of  $C\_out\_u$ . On receiving a message sent along  $c_{p,q}^v$ , from any one *in\_queue* in  $C\_in\_v$ , *sp* places its copies in all the *out\_queues* in  $C\_out\_v$  and discards all its copies received later in other *in\_queues*. This method of sending messages along a subset of outgoing channels is called *selective flooding*.

Thus, for  $u \leq v$  and two messages  $m_i$  and  $m_j$ , such that  $m_j$  is sent along  $c_{p,q}^v$  after sending  $m_i$  along  $c_{p,q}^u$ , a copy of  $m_i$  will be ahead of  $m_j$  and  $m_j$  will never overtake  $m_i$ . Hence,  $c_{p,q}^u$  will be at least as fast as  $c_{p,q}^v$ . Moreover, let  $v < w$ , and  $m_j$  be sent along  $c_{p,q}^v$  after sending  $m_i$  along  $c_{p,q}^w$ . If there is at least one path in  $\mathcal{C}^v - \mathcal{C}^w$ , then along any such path or an *in\_queue* in  $C\_in\_v - C\_in\_w$  message  $m_j$  may be received by *sp* at a switch node (including the sink node) before  $m_i$ . So,  $m_j$  may overtake  $m_i$ . Thus, the level property is satisfied.

#### 4.5 Data Structures and Algorithm for *sp* at a Switch Process

For simplicity we assume that initially each level  $v$  has been statically assigned a subset of paths in the *s-net*. So  $C\_in\_v$  and  $C\_out\_v$  are known in advance. A message at level  $v$  is said to have been sent by *sp* if its copies have been put in each *out\_queue* in  $C\_out\_v$ . A message at level  $v$  is said to have been received by *sp* when its copy has been received by *sp* from at least one *in\_queue* in  $C\_in\_v$ .

After receiving the first copy of a message, *sp* discards all other copies subsequently received.

To do so, *sp* maintains a linked list, called *Ignore\_list*.

---

such *s\_channel* may be implemented using any other implementation of level property [17] or recursively using the algorithm presented here.

For each level at which copies of message(s) to be discarded are expected, there is a record in the list. The *in\_queues* and *out\_queues* of *sp* are numbered starting from 0. Space for a record corresponding to level *u* is allocated at *sp* when *sp* receives a message at that level and the following two conditions are satisfied:

1.  $|C_{in-u}| > 1$
2. there is no record for level *u* in the linked list

Each record has three fields:

1. *level*: a real valued variable
2. *count*: an array of as many integers as the number of *in\_queues* incident on *sp*
3. *next\_ptr*: a pointer to the next record in the list.

The array *count* is used to keep track of the number of messages to be received along each *in\_queue* in the future, that have to be discarded.

On receiving a message *m*, at level *u*, from an *in\_queue* *i*, *sp* invokes the function *find* to search *Ignore\_list* for the record corresponding to level *u*. If the search is successful *find* returns a pointer *m\_id* to the record. Otherwise, it returns NULL.

If  $m\_id \uparrow .count[i]$  is greater than zero then *m* is a copy of a message that has already been forwarded by *sp* to the next switch node on its way to its destination. So, it is discarded and  $m\_id \uparrow .count[i]$  is decremented by one. If as a result of this decrease all the count fields in the record for level *u* become zero the record is deleted from *Ignore\_list*.

Otherwise, if  $m\_id \uparrow .count[i]$  field is zero, it means that *sp* has not received a copy of *m* before. So, a copy of *m* is placed in each *out\_queue* in *C\_out\_u*. If  $|C_{in-u}| > 1$  then  $m\_id \uparrow .count[j]$  is incremented by one for each *in\_queue*  $j \neq i$  in *C\_in\_u* along which copies of *m* (to be discarded later) are expected in the future.

If *find* returns NULL then a record is created for level *u* provided the two conditions specified earlier are satisfied. All the *count[i]* fields are initialized to zero and then the operations corresponding to the previous paragraph are executed.

```

message-record = record
    level : real;
    count : array[total number of in_queues at sp] of integer;
    next-ptr :  $\uparrow$  message-record;
end.

```

```

Ignore_list, m_id :  $\uparrow$ message-record; (both initially NULL)
u : integer;

```

**Algorithm executed at *sp* ::**

```

*[Non-deterministically select any non-empty in_queue i and receive a message m from its head;
u  $\leftarrow$  level of m;
m_id  $\leftarrow$  find(Ignore_list, u);
if (m_id  $\neq$  NULL)then
    {if (m_id $\uparrow$ .count[i]>0) then
        {m_id $\uparrow$ .count[i] $\leftarrow$ m_id $\uparrow$ .count[i]-1
        if m_id $\uparrow$ .count[j]=0  $\forall j$  then delete(Ignore_list, m_id);
        }
    }
    else/* a new message has been received */
        forward_new_message(m, m_id, u, i);
else/* no record for level u in list */
    {add record for level u in Ignore_list with m_id pointing to it;
     $\forall i \in C_{in\_u}$  m_id $\uparrow$ .count[i] $\leftarrow$ 0;
    forward_new_message(m, m_id, u, i);
    }
}
]

```

**forward\_new\_message(m, m\_id, u, i)**

```

{put a copy of m in all out_queues in  $C_{out\_u}$ 
if  $|C_{in\_u}| > 1$  then
     $\forall in\_queues j \in C_{in\_u}$  ( $j \neq i$ ) m_id $\uparrow$ .count[j] := m_id $\uparrow$ .count[j]+1;
}

```

## 4.6 Proof of Correctness of the Algorithm

**Lemma 1** : *No messages are dropped and no spurious messages are generated by the algorithm.*

**Proof:** In order to prove the lemma we need to prove the following assertions:

**Assertion 1:** When the first copy of a level *u* message *m* is received by the switch process *sp*  $m\_id = NULL \vee m\_id \uparrow .count[i] = 0$ , where *i* is the queue in which the first copy of *m* is received.

*Proof by induction on the number of messages received by sp at level u.*

*Base Case:* When the first copy of the first level *u* message, *m*, is received by *sp* then  $m\_id = NULL \vee m\_id \uparrow .count[i] = 0$

This proceeds from the fact that *Ignore\_List* at a switch process is initialized to NULL. Hence,

when  $m$  arrives at  $sp$ ,  $find(Ignore\_list, u)$  returns  $NULL$ . So,  $m\_id = NULL$ .

Induction Hypothesis: Let the assertion hold when the first copy of message number  $N$  at level  $u$  (not a copy of a message that has already been received along other  $in\_queues$ ) is received by  $sp$ .

Induction Step: We prove that the assertion holds when the first copy of message number  $N + 1$  at level  $u$  is received by  $sp$ .

Let the first copy of the  $N + 1^{st}$  message  $m'$  arrive at  $sp$  along  $in\_queue$   $i$ . There are two possibilities with respect to the  $N^{th}$  message  $m$ :

1. The first copy of message  $m$  arrived at  $sp$  along  $in\_queue$   $i$ . Therefore  $m\_id = NULL \vee m\_id \uparrow .count[i] = 0$  when  $m$  arrived along  $i$ . If  $m\_id \uparrow .count[i] = 0$  then it implies that:

- $m\_id \uparrow .count[i]$  is not incremented when copies of  $m$  are forwarded by  $sp$  towards the destination.
- $m\_id \uparrow .count[i]$  is equal to zero when the first copy of  $m'$  arrives at  $sp$  along  $in\_queue$   $i$ . This is because  $m\_id \uparrow .count[i]$  is updated only on the arrival of a level  $u$  message along an  $in\_queue$ , and it was not incremented when the last message,  $m$ , was received along  $in\_queue$   $i$ .

If  $m\_id = NULL$  when the first copy of  $m$  arrived at  $sp$  along  $i$ , and  $C\_in\_u - \{i\} \neq \phi$ , then a record corresponding to level  $u$ , with pointer  $m\_id$ , is created and  $m\_id \uparrow .count[i]$  is set to zero which is the value seen on the arrival of the next level  $u$  message  $m'$ . If  $C\_in\_u - \{i\} = \phi$ , then a record for level  $u$  is not created on the arrival of  $m$ . So,  $m\_id = NULL$  when the first copy of  $m'$  arrives at  $sp$ .

2. The first copy of message  $m$  arrived at  $sp$  along  $in\_queue$   $j$ , where  $i, j \in C\_in\_u$  and  $i \neq j$ . Therefore:

- On the arrival of  $m$ ,  $m\_id \uparrow .count[i]$  is incremented by one, because a copy of  $m$  is expected to arrive later along  $in\_queue$   $i$ .
- A copy of  $m$  is received at  $sp$  along  $in\_queue$   $i$  before the first copy of  $m'$  is received along  $i$ . This follows from the fact that messages at the same level exhibit *FIFO* order along the  $s\_channels$  and the queues are *FIFO* too. As  $m$  was sent before  $m'$ , at the same level,  $m$  cannot be overtaken by  $m'$ .

- As  $m\_id \uparrow .count[i]$  is greater than zero when a copy of  $m$  along  $i$  is received,  $m\_id \uparrow .count[i]$  is decremented by one. If all the count values in the record pointed to by  $m\_id$  become zero, the record is deleted.

Thus for each message among the first  $N$  level  $u$  messages received by  $sp$ ,  $m\_id \uparrow .count[i]$  is incremented by one if the first copy of the that message is received along some  $in\_queue j : j \neq i$ . A copy of every such message is received along  $in\_queue i$  before  $m'$ . Whenever such a copy is received along  $in\_queue i$ ,  $m\_id \uparrow .count[i]$  is decremented by one. Hence, if the first copy of  $m'$  arrives along  $in\_queue i$ , then on its arrival  $m\_id \uparrow .count[i] = 0 \vee m\_id = NULL$ .

**Assertion 2:** If  $m\_id \uparrow .count[i] = 0$  at  $sp$  when a copy of a message  $m$  is received along  $in\_queue i$ , then it must be the first copy of  $m$  to be received by  $sp$ .

*Proof by contradiction:*

Let us assume that the copy of message  $m$  received along  $in\_queue i$  when  $m\_id \uparrow .count[i] = 0$  is a duplicate copy, i.e., a copy of  $m$  has already been received at  $sp$  along some other  $in\_queue$ . According to the algorithm,  $m\_id \uparrow .count[i]$  is incremented by one when the first copy of a message at level  $u$  arrives at  $sp$  along an  $in\_queue j : j \neq i$ , and is decremented by one when a copy of a message arrives along  $in\_queue i$  and finds  $m\_id \uparrow .count[i] > 0$ . So, our assumption would imply that the copy of  $m$  received along  $in\_queue i$  has been overtaken by some other message(s) at the same level ( $u$ ) along its path from the source to  $sp$ . As the channels and queues are *FIFO* with respect to messages at the same level, this is not possible — a contradiction.

From the two assertions it can be concluded that  $m\_id \uparrow .count[i] = 0 \vee m\_id = NULL$  iff a new message arrives at  $sp$  along  $in\_queue i$ . According to the algorithm, copies of a message are propagated by the switch process towards the destination only when the condition mentioned above is satisfied. Thus, copies of a message are propagated by a switch process iff that message is not a copy of a message that has already been propagated. Thus no messages are dropped and no phantom messages are generated by intermediate switch nodes. ■

**Lemma 2 :** *Let a message  $m$  be sent by  $p$  to  $q$  along  $c_{p,q}^v$  after  $m'$  is sent by  $p$  to  $q$  along  $c_{p,q}^u$  such that  $u \leq v$ . Then  $m$  is received after  $m'$  at each switch node  $s$  that lies on both  $c_{p,q}^u$  and  $c_{p,q}^v$ .*

**Proof:** The proof is by induction on  $n$ , the number of switch nodes on the longest path between  $p$  and  $s$  at level  $v$ .

Base Case: Let  $n = 0$ . Then  $s$  must be the source switch node  $p$ . During the transmission of  $m'$ ,  $p$  puts a copy of the message in its *out\_queue* which is also the *in\_queue* in the *s-net* for  $c_{p,q}$ . This is always the *in\_queue* along which the first copy of  $m'$  is received by the switch process at  $s$ , as the other *in\_queues* at  $s$ , if any, arise due to the cyclicity of the *s-net* (these other *in\_queues* at  $s$  can receive a copy of the message only after it has been propagated by the switch process at  $s$ ). Later  $p$  puts a copy of  $m$  in its *out\_queue*. As the *out\_queue* of  $p$  is *FIFO*, the switch process at  $s$  can receive  $m$  only after it has received  $m'$ .

Induction Hypothesis: Let the Lemma hold for each  $s'$  such that there are no more than  $n(\geq 0)$  switch nodes on the longest level  $v$  path between  $p$  and the  $sp$  at  $s'$ .

Induction Step: Let  $s$  be a switch node that lies along both  $c_{p,q}^u$  and  $c_{p,q}^v$  such that there are  $n + 1$  switch nodes on the longest path (at level  $v$ ) between  $p$  and the  $sp$  at  $s$ . The paths corresponding to level  $v$  between  $p$  and  $s$  are a subset of the paths corresponding to level  $u$  between the same pair of nodes. So, for every path along which a copy of  $m$  is sent the following can be inferred:

- a copy of  $m'$  was sent along this path before sending a copy of  $m$ .
- this level  $v$  path is made up of a path from  $p$  to a switch node  $s'$  with  $n$  switch nodes on it and an *s\_channel* from  $s'$  to  $s$ .
- from the induction hypothesis, the Lemma holds at  $s'$ , i.e., if  $m$  is sent by  $p$  at level  $v$  after  $m'$  is sent at level  $u$  ( $u \leq v$ ) then  $m$  is received at  $s'$  after  $m'$ .
- $sp$  at  $s'$  sends  $m$  towards  $s$  after  $m'$  because, according to the algorithm, copies of a new message are propagated by  $sp$  towards the destination before another message is selected from any *in\_queue*.
- this is analogous to the source  $s'$  sending  $m$  to  $s$  along  $c_{p,q}^v$  after sending  $m'$  to  $s$  along  $c_{p,q}^u$  where the number of switch nodes on the longest path between  $s'$  and  $s$  is one. Applying the induction hypothesis to the path  $s's$ ,  $m$  will be received after  $m'$  by  $s$  along every level  $v$  path.

Thus proved. ■

**Lemma 3** : If  $m'$  is sent by  $p$  to  $q$  along  $c_{p,q}^u$  after  $m$  is sent by  $p$  to  $q$  along  $c_{p,q}^v$  ( $u \leq v$ ) then  $m'$  may overtake  $m$  at any switch node  $s$  that lies on both  $c_{p,q}^u$  and  $c_{p,q}^v$  and may arrive at  $q$  before  $m$ .

**Proof:** Once again the proof is by induction on  $n$ , the number of switch nodes on the longest path between  $p$  and  $s$  at level  $v$ . We have seen that when  $n = 0$  overtaking is not possible as queues are *FIFO*. So, we will use a different base case.

Base Case: Let  $n = 1$ . This implies that the corresponding path at level  $v$  is composed of one *s\_channel* between  $p$  and  $s$ . During the transmission of  $m$ ,  $p$  puts a copy of the message in its *out\_queue* which is the *in\_queue* for the switch process corresponding to  $c_{p,q}$ . This switch process takes the message and puts it in the *out\_queues* corresponding to  $C_{out_v}$  from where they move towards  $s$  which is at a distance of one *s\_channel*. Later, when  $p$  puts a copy of  $m'$  in its *out\_queue*, the switch process puts copies of  $m'$  in the *out\_queues* corresponding to  $C_{out_u}$ , such that  $C_{out_v} \subseteq C_{out_u}$ . The copies of both the messages ( $m$  and  $m'$ ) travel along their paths and reach the corresponding *in\_queues* at  $s$ . At  $s$ ,  $C_{in_v} \subseteq C_{in_u}$ . So, in each *in\_queue*  $\in C_{in_v}$  there is a copy of  $m$  followed by a copy of  $m'$ . But in each *in\_queue* in the set  $C_{in_u} - C_{in_v}$  there is a copy of  $m'$  but no copy of  $m$ . The *sp* at  $s$  non-deterministically selects an *in\_queue* and extracts the message at its head. If the *in\_queue* selected at  $s$  belongs to  $C_{in_v}$  then the arrival of  $m$  is recorded before the arrival of  $m'$ . If an *in\_queue* in  $C_{in_u} - C_{in_v}$  is selected before any *in\_queue* in the set  $C_{in_v}$  is selected, then the arrival of  $m'$  is recorded before the arrival of  $m$ , in which case  $m'$  is said to have overtaken  $m$ .

Induction Hypothesis: Let the Lemma hold for each  $s'$  such that there are no more than  $n(\geq 1)$  switch nodes on the longest path between the *sp* at  $s'$  and  $p$ , at level  $v$ .

Induction Step: Let  $s$  be a switch node that lies on  $c_{p,q}^u$  and  $c_{p,q}^v$  such that there are  $n + 1$  switch nodes on the longest path (at level  $v$ ) between  $p$  and the *sp* at  $s$ . The longest path of length  $n + 1$  is made up of a path from  $p$  to a switch node  $s'$  with  $n$  switch nodes on it and an *s\_channel* from  $s'$  to  $s$ . From the induction hypothesis it can be inferred that the Lemma holds at  $s'$ . There are two possibilities:

1.  $m'$  has overtaken  $m$  somewhere between  $p$  and  $s'$ . Then from Lemma 2,  $m$  cannot overtake  $m'$  between  $s'$  and  $s$ . Hence,  $m'$  arrives at  $s$  before  $m$ .
2.  $m'$  has not overtaken  $m$  between  $p$  and  $s'$ . So,  $s'$  sends  $m'$  to  $s$  after sending  $m$ . This is

similar to the base case where  $s'$  and  $s$  correspond to the source and destination, respectively, with  $n = 1$ . Hence,  $m'$  may overtake  $m$  between  $s'$  and  $s$ . ■

**Theorem 1** *The algorithm ensures the level property for  $c_{p,q}$ .*

**Proof:** Lemma 1 states that the algorithm does not generate any spurious messages and no messages sent by the source to the destination are dropped at any switch node. Lemma 2 states that the algorithm does not permit messages sent at higher levels to overtake messages sent at lower levels. Lemma 3 states that the algorithm permits messages sent at lower levels to overtake those sent at higher levels. Thus the algorithm ensures that the level property is satisfied for  $c_{p,q}$ . ■

In the proposed algorithm the possibility of a message waiting infinitely long at a switch node in an *in\_queue* is avoided by assuming that nondeterministic selection of any non-empty queue leads to the selection of each non-empty *in\_queue* in a finite time. Thus starvation does not take place.

So far we have assumed that on the *s-net* two adjacent switch nodes are connected by *FIFO s-channels*. This constraint can be relaxed so that the underlying *s-channels* themselves exhibit the level property. It is easy to see that a channel  $c_{p,q}$  composed of such paths will exhibit the level property. Similarly, the requirement of *FIFO* queues can be relaxed to queues that output a message in an order that satisfies the level property.

#### 4.7 Performance Analysis of Hierarchical Channels

Given the topology of the *u-net* and the volume of traffic, it is possible to make a probabilistic prediction of the time taken for messages to travel at a given level between a pair of switch nodes. The performance analysis accounts for the following factors:

- the different paths corresponding to a given level may be of different lengths, both in terms of the physical distance traveled and the number of intermediate nodes.
- the time to travel over a given length of physical medium can be determined fairly accurately.
- due to the interaction between different messages at the switch nodes, the delays involved in processing a message at a switch node can vary.

- the sum of the time taken to travel along the physical links and the delays at the nodes in the path from the source to the destination gives the total time for a message to travel along that path from the source to the destination.
- in the presence of multiple paths, the minimum travel time of a message copy, among all the paths from the source to the destination, gives us the time for the message to reach the destination.

Let there be a switch node  $q$  in the  $s$ -net between the source and the destination. There are  $n_q$  *in\_queues* in the  $u$ -net incident into  $q$ , of which  $x_{u,q}$  correspond to level  $u$  ( $x_{u,q} = |C_{in_u}|$  at switch node  $q$ ). Assuming that all the *in\_queues* are identical, the probability that any one of these  $x_{u,q}$  *in\_queues* will be selected by the  $sp$  at  $q$  to extract a message is equal to  $x_{u,q}/n_q$ .

Messages generated by nodes have to be propagated through the  $u$ -net. These messages arrive at the *in\_queues* of the corresponding  $s$ -nets from outside the network. At the destination nodes, the messages exit the network. Thus the whole system can be modeled as an open network of queues. We assume that the message generation process at node  $q$  is a Poisson process and the message service time at  $q$  is exponentially distributed with mean service rate  $\mu_q$ . So, the expected service rate for each *in\_queue* is equal to  $\mu_q/n_q$ . Message service time denotes the time spent by the switch node  $q$  in extracting the message from the head of its *in\_queue* and sending it towards the destination (by putting copies of it in the appropriate *out\_queues*). Assuming steady state behaviour, the internal arrival streams at the nodes (message copies traveling between intermediate nodes in the path from the source to the destination) have Poisson distribution. So, at node  $q$  the  $n_q$  *in\_queues* and the switch process  $sp$  with service rate  $\mu_q$  can be considered to be  $n_q$  identical M/M/1 systems, each with the arrival process having Poisson distribution with parameter  $\lambda_q$  and mean service rate of  $\mu_q/n_q$ . Thus the mean waiting time for a message at node  $q$ , in an *in\_queue*, before it reaches the head of the *in\_queue*, is given by the following expression:

$$\frac{\lambda_q n_q^2}{\mu_q (\mu_q - \lambda_q n_q)}$$

A message is said to have been processed by a switch node as soon as its first copy has been extracted by the switch node and forwarded towards the destination. Assuming that at a particular instant of time there are  $x$  copies of a message at the heads of different *in\_queues* at switch node  $q$ ,

the expected number of selections required for at least one of these *in\_queues* to be chosen is equal to:

$$\sum_{i=1}^{\infty} i(1 - x/n_q)^{i-1} x/n_q = n_q/x$$

Each of these selections will take  $1/\mu_q$  units of time to be processed. Therefore, the expected time for the message to be processed in such a situation is equal to:  $n_q/(x\mu_q)$ .

For a message at level  $u$ , the value of  $x$  varies from 1 to  $x_{u,q}$ , each with a corresponding probability of  $p_x$ . The value of  $p_x$  depends on factors like the current traffic on the network, the topology of the  $u$ -net, etc. So,  $p_x$  is difficult to calculate analytically. But, it can be determined experimentally by collecting statistics about the behaviour of the network. Thus, the expected service time of a level  $u$  message at node  $q$  is equal to  $\sum_{x=1}^{x_{u,q}} (n_q p_x)/(x\mu_q)$ . The variance of this distribution also depends on  $p_x$ .

Therefore the expected total time to process a level  $u$  message at a switch node  $q$  is equal to:

$$T_q = \frac{\lambda_q n_q^2}{\mu_q(\mu_q - \lambda_q n_q)} + \sum_{x=1}^{x_{u,q}} \frac{n_q p_x}{x\mu_q}$$

The second part in the expression for  $T_q$  can be written as:  $(n_q/\mu_q) \sum_{x=1}^{x_{u,q}} (p_x/x)$ , where  $\sum_{x=1}^{x_{u,q}} p_x = 1$ . So, as the value of  $x_{u,q}$  increases, the value of  $\sum_{x=1}^{x_{u,q}} (p_x/x)$  decreases. Hence, the lower the level ( $u$ ) of a message, greater the value of  $x_{u,q}$ , and smaller the time to process the message at at switch node  $q$ .

For a path from the source to the destination consisting of  $k$  switch nodes, the time spent by a level  $u$  message at the switch nodes is equal to  $T' = \sum_{q=1}^k T_q$ . The sum of  $T'$  and the time taken to travel along all the *s\_channels* in a path from the source to the destination is the elapsed time between a level  $u$  message leaving the source and arriving at the destination along that path. If there are several disjoint paths in the  $s$ -net corresponding to level  $u$  then the minimum elapsed time ( $T_{min}$ ) among all the paths is the expected time taken by a level  $u$  message.

Our analysis is conservative. It assumes that the duplicate copies of messages that have already been propagated by the switch process are also processed on arrival. Hence, our analysis assumes a higher volume of traffic and a lower service rate than will actually arise in the algorithm. So, the results presented above give an upper bound on the expected time for a level  $u$  message to travel from the source to the destination.

## 4.8 Recovery from $s\_channel$ Failures

As explained earlier, there may be several paths between two nodes  $p$  and  $q$  in the  $s\_net$   $c_{p,q}$ . Some of these paths may consist of direct  $s\_channels$  between  $p$  and  $q$  while some other paths may have intermediate nodes between them. Figure 4 shows an example  $s\_net$  where the switch processes are represented by circles and the  $s\_channels$  are shown by arrows.

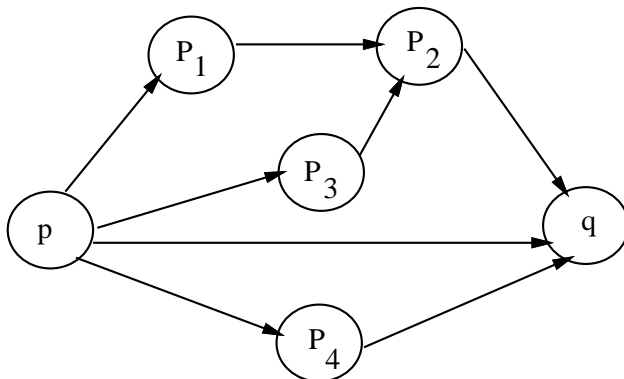


Figure 4: An example  $s\_net$ .

It is possible that some of the  $s\_channels$  may temporarily develop faults and be unable to deliver messages. Such faults may lead to the violation of the level property as explained below with the help of Figure 4.

Let us assume that a message  $m$ , at level  $u$ , is sent by  $p$  to  $q$ , at time  $t$ . The paths taken by the copies of the message are  $pP_1P_2q$ ,  $pq$  and  $pP_4q$ . At the time  $m$  is sent, the paths  $pq$  and  $pP_4q$  are out of order due to one or more  $s\_channels$  along these paths being faulty. So, only one copy of  $m$  can be delivered to  $q$  along the path  $pP_1P_2q$ . Later, at time  $t'$ , another message  $m'$  is sent by  $p$  to  $q$  at level  $v$ , where  $u < v$ . The paths taken by  $m'$  are  $pP_1P_2q$  and  $pq$ , both of which are fault-free at that time. There is a possibility that the copy of  $m'$  sent along  $pq$  may reach  $q$  before the copy of  $m$  sent along  $pP_1P_2q$  does, thus violating the level property.

We only consider channel failures that are fail-stop in nature. Our algorithm handles only those channel failures where there is at least one operational path in the  $s\_net$  between the source and the destination for the duration of the failure. The two nodes, on which the faulty  $s\_channel$  is incident, get to know of the failure before they try to send or receive any more messages along that channel. We will call the switch process that sends messages on an  $s\_channel$  as the *head* of the  $s\_channel$  and the switch process that receives messages on the  $s\_channel$  as the *tail* of the

*s\_channel*. At a switch node a sequence number is associated with each level. It is initialized to zero. Each time a message at a given level is sent by the *head*, the corresponding sequence number is incremented by one. For an outgoing faulty *s\_channel*  $c$  the sequence number of the last message sent at level  $u$  before its failure is stored by  $head(c)$  in  $low\_num_u(c)$ . So, messages with higher sequence numbers cannot be delivered until  $c$  is repaired.

At  $head(c)$  it is not practical to buffer all the messages that could not be sent along  $c$  while it was out of service. Buffering would require a large amount of memory. So, each time a message at level  $u$  has to be sent out along  $c$  when it is out of service,  $revive\_link_u(c)$  (initialized to zero at the time of  $c$ 's failure) is incremented by one and the message is discarded. When  $c$  has been repaired a recovery phase is started.

### Recovery Phase

In the beginning of the recovery phase  $head(c)$  sends  $revive\_link_u(c)$  and  $low\_num_u(c)$  to  $tail(c)$ , along  $c$ . Let the sequence number of the latest message at level  $u$  received by  $tail(c)$  from the source, along any other *s\_channel*, when  $revive\_link_u(c)$  is received, be  $high\_num_u(c)$ . Hence, the upper bound on the number of messages at level  $u$  arriving at  $tail(c)$ , along  $c$ , that should be discarded is equal to  $high\_num_u(c) - low\_num_u(c)$ . This value is stored in  $receive\_link_u(c)$ . On receiving the message,  $tail(c)$  stores the value of  $receive\_link_u(c) - revive\_link_u(c)$  in  $ignore_u(c)$ .

Depending on the topology of the *s-net* there are two possibilities:

1. There is at least one operational path in the *s-net*, between the source of the messages and  $tail(c)$ , that does not contain  $c$ . In such a situation, let  $m$  be the next message received by  $tail(c)$  along  $c$  at level  $u$ . There are three possible cases:
  - (a)  $ignore_u(c) > 0$ : This implies that the next  $ignore_u(c)$  messages at level  $u$  arriving at  $tail(c)$  along  $c$  have already been received along other paths and should be treated as duplicate copies. Therefore,  $m$  is discarded and  $ignore_u(c)$  is decremented by one.
  - (b)  $ignore_u(c) = 0$ : This implies that for every message at level  $u$  that was delivered to  $tail(c)$  during  $c$ 's failure, a copy of the message was discarded at  $head(c)$  or has been delivered along  $c$  to  $tail(c)$ . So, if for all  $v \neq u$ ,  $u < v \vee ignore_v(c) \geq 0$ , then the message  $m$  arriving along  $c$  is treated as a regular message arriving along a fault-free *s\_channel*

would be. So, if this is the first copy of  $m$  received by  $tail(c)$  it is forwarded towards the destination. If it is a duplicate then it is discarded. If the constraint mentioned above is not satisfied, then selection of messages from the  $in\_queue$  corresponding to  $c$  is suspended until the constraint can be satisfied.

- (c)  $ignore_u(c) < 0$ : This implies that the sequence number of  $m$  is higher than the sequence number of the latest level  $u$  message received by  $tail(c)$  along any other  $s\_channel$ . Actually, the sequence number of the latest message received by  $tail(c)$  along any other  $s\_channel$  is equal to  $x + ignore_u(c) - 1$ .

So, if  $m$  were to be accepted by  $tail(c)$  the *FIFO* ordering among messages corresponding to the same level would be violated. To maintain the *FIFO* order among message at the same level  $|ignore_u(c)|$  new messages at level  $u$  with sequence numbers less than that of  $m$  should first be accepted by  $tail(c)$  from  $s\_channels$  other than  $c$ . The value of  $ignore_u(c)$  is incremented by one each time such a message arrives at  $tail(c)$  along some  $s\_channel$  other than  $c$ .

Hence, the selection of  $m$  from the *head* of the  $in\_queue$  for  $s\_channel$   $c$  is suspended until  $ignore_u(c)$  becomes zero. At that stage, action corresponding to the case when  $ignore_u(c) = 0$  is performed and  $m$  is processed.

2. The only path in the  $s\_net$  at level  $u$  from the source to  $tail(c)$  is through  $c$ . So, the number of messages at level  $u$  received by  $tail(c)$  is always less than or equal to those received by  $head(c)$ , and  $ignore_u(c)$  can either be equal to or less than zero.

If  $ignore_u(c) < 0$  for any  $u$ , then for every such  $u$ ,  $tail(c)$  sends an integer  $last_u$  in a special control message along all the outgoing  $s\_channels$ ,  $c'$  in the  $s\_net$ , corresponding to level  $u$ . The value of  $last_u$  is equal to  $low\_num_u(c) + revive\_link_u(c)$ . Having sent the control message,  $tail(c)$  sets its  $high\_num_u(c)$  value to be equal to  $last_u$  and handles every subsequent message as if  $c$  had never been faulty. When  $tail(c')$  receives  $last_u$ , it subtracts this value from  $high\_num_u(c')$  and stores the value in  $ignore_u(c')$ . If  $ignore_u(c') \neq 0$  for any  $u$  then  $tail(c')$  behaves as if  $c'$  had been faulty before receiving the control message and follows the strategy described above for  $tail(c)$ .

The recovery phase at a switch node  $c$  terminates when  $ignore_u(c) = 0$  for all  $u$ .

**Lemma 4** *The algorithm for recovery from  $s\_channel$  failures ensures that the level property holds at all the switch nodes in the  $s$ -net.*

**Proof:** We have already assumed that whenever an  $s\_channel$  fails there is at least one operational path in the  $s$ -net between the source and the destination for the duration of the failure for the levels at which messages are sent in that duration. Hence, no messages are lost during failures.

After an  $s\_channel$  has been repaired, a message  $m$  at level  $u$  is processed iff  $ignore_u(c) = 0 \wedge (\forall v \neq u : u < v \vee ignore_v(c) \geq 0)$ . The condition  $ignore_u(c) = 0$  guarantees that the messages at the same level are delivered in *FIFO* order.

If  $v \neq u \wedge u < v$  when a message  $m$  at message at level  $u$  is processed, the level property is not violated. This is because messages  $m'$  sent by the source at level  $v$  before  $m$  can be overtaken by the latter as it is at a lower level than the former.

If  $ignore_v(c) \geq 0$  at  $tail(c)$ , then there are some messages corresponding to level  $v$  sent by the source before  $m$ , which have already been received by  $tail(c)$ . But their copies have not been received by  $tail(c)$  along  $s\_channel$   $c$ . If  $u < v$  then the level property is not violated when  $m$  is processed by  $tail(c)$ . If  $u > v$ , and  $m$  is processed by  $tail(c)$  then it may appear that a message at a higher level has overtaken a message at a lower level along  $s\_channel$   $c$  and the level property has been violated. But, copies of these overtaken messages  $m'$  at level  $v$  have already been received by  $tail(c)$  and forwarded towards the destination. So, in effect the arrival of  $m$  is recorded by  $tail(c)$  only after the arrival of  $m'$ . As the path from  $tail(c)$  to the destination satisfies the level property  $m$  cannot overtake  $m'$  between  $tail(c)$  and the destination. Thus the level property is satisfied. ■

## 5 Implementation of Statistical Channels

The algorithm presented in Section 4.5 has different performance characteristics from a counter based implementation [17]. Implementations based on the proposed algorithm may increase message traffic while the counter based implementation may increase delays. Thus the proposed algorithm has uses whenever an increase in message traffic is acceptable over delays. In addition to these applications, the proposed algorithm has another special application resulting from the use of a set of paths and the selective flooding.

Statistical channels are characterized as follows: *the probability that the propagation delay of a message along the channel is smaller than  $D$  is at least  $P$*  [6]. Higher  $P$  denotes strong constraints on the channel performance, while lower  $P$  denotes a greater amount of flexibility. Hence,  $P$  is sometimes called the flexibility factor. Clearly, such channels have many uses which include offering real-time guarantees with probability  $P$ .

## 5.1 Implementation using Hierarchical Channels

In several multimedia applications when a source wants to communicate with the destination it specifies the desired channel characteristics in statistical terms: the peak and average bandwidth for the communication as well as the time bound. In this section we describe how selective flooding can be employed to establish statistical channels. We propose a new method called *iterative negotiation* which is explained below. We have used some of the symbols and notation of [6].

Let the time bound ( $D_j$ ) and the flexibility factor ( $P_j$ ) for a communication session  $j$  between the source and the destination be given. Based on these specifications and the observed volume of traffic in the network the session is tentatively mapped to a level  $u$  hierarchical channel. This hierarchical channel will also be identified with the letter  $j$ . The *s\_channel* delays along the path(s) corresponding to level  $u$  are subtracted from  $D_j$  and the balance, along with  $P_j$  is subdivided among the switch nodes on the hierarchical channel. Let the delay bound and probability thus assigned to switch node  $n$  be  $d_{j,n}$  and  $p_{j,n}$ , respectively. Let the probability of a message being delayed beyond its delay bound (deadline overflow) at  $n$  be  $P_{do,n}$ . An *overflow combination* is defined as a set of hierarchical channels, incident on a switch node, that when active concurrently for an extended period of time may cause messages to miss deadlines.  $P_{do,n}$  is the sum of the probabilities of all the overflow combinations that are possible.

The statistical tests described in Section 4.7 are used to determine if for each hierarchical channel  $j$  passing through  $n$ , the probability of delay higher than the bound  $d_{j,n}$  is below the maximum tolerable value,  $1 - p_{j,n}$ . This can be determined by checking if the following expression is satisfied [6]:

$$P_{do,n} \leq \min(1 - p_{j,n})$$

The statistical channel can be established using level  $u$  hierarchical channels only when the following two conditions are satisfied:

1. the probability of the expected time being higher than the delay bound is less than the maximum tolerable value.
2. the establishment of this statistical channel will not raise the probability of the expected delays of other message channels, which share some of the *s\_channels* that would be used to establish this channel, exceeding their delay bounds beyond their respective maximum tolerable values.

If the conditions are satisfied, appropriate resources are allocated at the intermediate nodes in the path and the channel is established. Otherwise, yet another attempt is made to establish the statistical channel with a higher priority, employing a level  $v$  hierarchical channel, where  $v < u$ . This process is repeated until the hierarchical channel so established satisfies the specification of the statistical channel, or it is determined that no hierarchical channel, satisfying the specification, can be established. Thus, the application negotiates with the *u-net* for a communication channel with the required quality of service. Clearly, the earliest time a message arrives at the destination (and we assume it is delivered instantaneously), will reduce as the number of paths over which it can travel increases.

## 6 Comparison with other Communication Protocols

*F-channels* cannot model all the possible message orderings, from strictly *FIFO* on one end to non-*FIFO* on the other. But, hierarchical channels, with their ability to send messages at different levels, offer greater flexibility. Also, hierarchical channels can prioritize messages by assigning different levels to them. Hence, we compare them with expedited data transfer protocols and priority queues.

### 6.1 Comparison with F-channels

Messages sent along *hierarchical channels* can easily simulate *F-channels*. An  $f$  message (forward flush message) cannot overtake any other message. This can be achieved by sending the message, which otherwise would have been sent as an  $f$  message, at a level higher than or equal to any other message previously sent. As a result it will not be able to overtake any message and subsequent messages (if sent at a lower level) can overtake it.

No message can overtake a  $b$  message (backward flush message). This can be achieved by ensuring that after sending the message, that otherwise would have been sent as a  $b$  message, all the subsequent messages are sent at the same or a higher level as compared to the message.

A  $t$  message (two way flush message) cannot overtake any message and no message can overtake it. This behavior can be simulated by sending the message, that otherwise would have been sent as a  $t$  message, at a level higher than or equal to that of any previous message and by ensuring that all the subsequent messages are sent at the same or a higher level as compared to the message.

The advantage of *hierarchical channels* over *F-channels* is that the former offer greater flexibility. If we refer back to Section 3.2, we find that *F-channels* permit fewer alternatives than *hierarchical channels*. For example, if  $a$  and  $b$  are sent as ordinary messages, they can be received in either order  $(a, b)$  or  $(b, a)$ . Variables  $c$  and  $d$  should both be received either before or after both  $a$  and  $b$  for  $p$ 's code to evaluate the expression correctly. Let  $q$  send  $c$  as an  $f$  message. Then  $c$  will be received after both  $a$  and  $b$ . But having done that *F-channels* do not allow us the freedom of sending  $d$  in such a fashion that it is received after  $a$  and  $b$  but either before or after  $c$ . The only permissible options are for  $d$  to be sent as an  $f$  message or a  $t$  message. Hence, the only alternative forms of evaluation permitted by *F-channels* are:  $(a + b) * (c + d)$  and  $(b + a) * (c + d)$ , i.e., only two out of the eight possible. Alternatively, if  $a$  is sent as an ordinary message,  $b$  as a  $t$  message, and  $c$  and  $d$  as ordinary messages in that order then the forms of evaluation permitted are  $(a + b) * (c + d)$  and  $(a + b) * (d + c)$ : again two out of the eight possible. For other possible combinations of flush messages that ensure correct evaluation of the expression by  $p$  and  $q$ , too, only two out of the eight possible alternatives are permitted. This example demonstrates that *F-channels* are more restrictive than *hierarchical channels*.

One important feature of simulating an *F-channel* using a hierarchical channel is that such simulations for  $b$ - or  $t$ -messages can be stopped any time, thus effectively relaxing restrictions. We illustrate this for a  $b$ -message which is useful for various algorithms in database management. Using the ability to change a message type, by stopping the simulation of  $b$ -message, as provided by *hierarchical channel*, but not by *F-channel*, certain algorithms that require such a change can be designed optimistically. In a hierarchical channel once a message  $m_i$  intended to simulate a  $b$ -message (at the time of sending) has been sent, later a message  $m_j$  at a lower level can be sent which has a chance of overtaking the  $m_i$ . If  $m_i$  carries information about an operation which needs

to be undone,  $m_j$  may be sent with the preemptive information and act as an anti-message [11]. So if it is received before the  $m_i$ , the latter is simply discarded when it reaches the destination.

## 6.2 Comparison with Expedited Data Transfer and Priority Queues

In a computer network the Network Layer and the Sessions Layer may provide primitives for *expedited data transfer* [21]. We will henceforth refer to packets of ordinary messages as *messages* and expedited packets as *expedited messages*. When an expedited message arrives at a node it is expedited using two level *priority queues*; one with regular priority and the other with expedited/higher priority.

There are two major differences between expedited data transfer and communication using hierarchical channels. First, expedited data transfer protocols make an explicit claim that messages sent using the protocol will overtake regular messages, while hierarchical channels claim that messages sent at a lower level are at least as fast as those at higher levels. So, a message sent at a higher level can never overtake a message sent at a lower level, but the converse is *possible*. This nondeterminism inherent in hierarchical channels provides flexibility as manifested by the examples in Section 3. This brings us to the other major difference. Unlike expedited data transfer protocols, hierarchical channels provide message transmission at several levels, providing even more flexibility.

*Priority queues* indicate how the queues behave, *i.e.*, how a message with higher priority overtakes one with a lower priority in the queue. In the case of *hierarchical channels*, let  $m_i$  be sent at level  $u$  after  $m_j$  is sent at level  $v$ , such that  $u < v$ . Then if  $m_i$  arrives at an *in\_queue* and  $m_j$  is already in that *in\_queue*, then  $m_i$  will be put *after*  $m_j$  in the *in\_queue*. This does not require any complicated queue manipulation. Now, for *priority queues*, let  $m_i$  be sent at a higher priority than  $m_j$ . Then irrespective of the order of sending, if  $m_i$  arrives at a queue and  $m_j$  is already in the queue then  $m_i$  will be placed *ahead of*  $m_j$  in the queue. This would require more than a simple enqueue operation.

For a network with only one path from the source to the destination, *priority queues* perform in accordance with the *hierarchical channel* specifications. But if there are multiple paths, then simply using *priority queues* will not enforce *hierarchical channel* specifications. For example, if  $m_i$  is sent before and at a lower level than  $m_j$ , and the two messages take entirely different paths, then there is a possibility that  $m_j$  reaches the destination before  $m_i$ , even though according to

*hierarchical channel* specifications  $m_j$  cannot overtake  $m_i$ .

The example described above illustrates the strength of hierarchical channels over priority queues. Priority queues only describe the behavior of message send and receive queues at nodes in the path between the source and the destination nodes. They make no assertions about message ordering along the links of the underlying network, or about end-to-end message ordering constraints. In contrast, hierarchical channels enable us to specify the end-to-end message ordering constraints. Thus, using hierarchical channels programmers can specify the semantics of their communication primitives. The implementation of hierarchical channels will ensure that message propagation along the links and send/receive queues in the underlying network is consistent with the ordering constraint specified by the communication primitives.

Thus there are two advantages of *hierarchical channels* over *priority queues* and *expedited data transfer*:

1. simplicity of queue management in *hierarchical channels*,
2. safety property enforcement in *hierarchical channels*. For example, a system message sent at a lower level cannot be overtaken by an application message at a higher level, thus permitting an interpretation of the application message as per the specification given by the system message.

Hierarchical channels can provide expedited data transfer by sending the corresponding messages at the lowest level. Flooding is employed for such messages, i.e., as soon as the switching process,  $sp$ , receives a copy of the message it forwards it along all the outgoing channels. The appendix presents such an implementation of *hierarchical channels* that also employs *priority queues* for expedited messages.

## 7 Conclusion

Several emerging networking and distributed computing applications do not require *FIFO* order of message delivery. Instead, they can operate with relaxed ordering constraints as long as certain quality of service criteria are satisfied. Hierarchical channels are suitable for the design and implementation of such applications as illustrated by the examples presented in the paper. Hierarchical

channels can support a wide range of message ordering constraints, right from *FIFO* to completely unordered message delivery.

Since *hierarchical channels* use just one parameter, level number, they are easy to use and implement. The overhead of the implementation strategy presented in this paper, using selective flooding, is increased message traffic. A previous counter based implementation strategy will avoid the increase in traffic but will lead to delays in message delivery [17, 18, 19]. Hence, the proposed strategy is suitable for environments in which an increase in the number of messages in the underlying network is preferable to an increase in the delays involved in message delivery. It may be possible to implement hierarchical channels using other, as yet unexplored, strategies. Some performance gains due to hierarchical channels will be manifested regardless of the underlying implementation of such channels, while some other benefits of hierarchical channels are specific to their implementation strategy. The choice of the implementation strategy for hierarchical channels will be guided by the relative costs of various resources required by the different implementation strategies.

An advantage of the selective flooding based strategy proposed in this paper is the ability to predict the expected message delays. This ability to predict the performance of hierarchical channels is especially useful for some multimedia applications, for example video-conferencing, that impose real-time constraints on the data streams. *Deterministic channels* with fixed delays could be used for such applications. But, it is difficult and expensive to support such channels. Instead, messages can be sent along hierarchical channels at appropriate levels to satisfy the quality of service requirements. Low level messages are expected to reach the destination process sooner than high level messages.

Hierarchical channels can also be made resilient to link-failures. They provide greater flexibility than *F-channels*, which in turn are more flexible than *FIFO* channels. Hierarchical channels also require simpler queue management than priority queues, and provide stronger guarantees about message ordering characteristics than priority queues.

The flexibility of hierarchical channels can be exploited to relax FIFO message ordering constraints in distributed applications. This leads to increased concurrency and improved performance.

## References

- [1] M. Ahuja. Flush primitives for asynchronous distributed system. *Information Processing Letters*, 34(1):5–12, February 1990.
- [2] M. Ahuja. Hierarchy of Communication Speeds for Designing Concurrent Systems. Technical Report OSU–CISRC–1/91–TR1, The Ohio State University, 1991.
- [3] M. Ahuja. An implementation of F-channels. *IEEE Transactions on Parallel and Distributed Systems*, pages 658–667, June 1993.
- [4] M. Ahuja and R. Prakash. On the relative speed of messages and hierarchical channels. In *Proceedings of the 4<sup>th</sup> IEEE Symposium on Parallel and Distributed Processing*, pages 246–253. IEEE Computer Society Press, 1992.
- [5] K. M. Chandy and L. Lamport. Distributed Snapshots : Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.
- [6] D. Ferrari and D. C. Verma. A Scheme for Real-Time Channel Establishment in Wide-Area Networks. *IEEE Journal on Selected Areas in Communications*, pages 368–379, April 1990.
- [7] J.-M. Helary. Observing Global States of Asynchronous Distributed Applications. In J-C Bermond and M. Raynal, editors, *Proc. of the 3<sup>rd</sup> International Workshop on Distributed Algorithms*. Springer-Verlag LNCS 392, 1989.
- [8] C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, August 1978.
- [9] T. Houdoin and D. Bonjour. ATM and AAL Layer Issues Concerning Multimedia Applications. *Annals of Telecommunications*, 49(5):230–240, 1994.
- [10] International Organization for Standardization. *ISO Open System Interconnection – Basic Reference Model ISO/TC 97/SC 16N719*, 1981.
- [11] D. R. Jefferson. Virtual Time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, July 1985.
- [12] R. Koo and S. Toueg. Checkpointing and Rollback-Recovery for Distributed Systems. *IEEE Transactions on Software Engineering*, SE-13(1):23–31, January 1987.
- [13] T.-H. Lai and T.-H. Yang. On Distributed Snapshots. *Information Processing Letters*, 25:153–158, 1987.
- [14] F. Mattern. Algorithms for Distributed Termination Detection. *Distributed Computing*, 2:161–175, 1987.
- [15] F. Mattern. Experience with a New Termination Detection Algorithm. In J. Van Leeuwen, editor, *Proc. of the 2<sup>nd</sup> International Workshop on Distributed Algorithms*, pages 127–143. Springer-Verlag LNCS 312, 1988.
- [16] R. Prakash and M. Singhal. Low-Cost Checkpointing and Failure Recovery in Mobile Computing Systems. *IEEE Transactions on Parallel and Distributed Systems*, pages 1035–1048, October 1996.

- [17] K. Shafer and M. Ahuja. Process-*Channel<sub>agent</sub>*-Process Model of Asynchronous Distributed Communication. In *Proceedings of International Conference on Distributed Computing Systems*, 1992.
- [18] K. Shafer and M. Ahuja. Distributed modeling and implementation of high performance communication architectures. In *proceedings Thirteenth Int. Conf. on Distributed Computing Systems*. IEEE, 1993.
- [19] K. Shafer and M. Ahuja. Implementation of hierarchical F-channels for high-performance distributed computing. *Distributed Computing (Springer-Verlag)*, 8(4):211–218, 1995.
- [20] R. E. Strom and S. Yemini. Optimistic Recovery in Distributed Systems. *ACM Transactions on Computer Systems*, 3(3):204–226, August 1985.
- [21] A. Tanenbaum. *Computer Networks(2<sup>nd</sup> Edition)*. Prentice Hall, Englewood Cliffs, N.J., 1988.
- [22] K. Taylor. The Role of Inhibition in Asynchronous Consistent-cut Protocols. In *Proceedings of the third International Workshop on Distributed Algorithms*, pages 280–291. Springer-Verlag LNCS 392, 1989.
- [23] S. Venkatesan. Message-optimal incremental snapshots. In *proceedings Ninth Inter. Conf. on Distributed Computing Systems*, pages 53–60. IEEE, 1989.
- [24] K. Venkatesh, T. Radhakrishnan, and H.F. Li. Global state detection in non-FIFO networks. In *Proceedings of the 7<sup>th</sup> International Conference on Distributed Computing Systems*, pages 364–370, 1987.
- [25] I. Wakeman. Packetized Video: options for interaction between the user, the network and the codec. *The Computer Journal*, 1993.

## A Implementation of Expedited Data Transfer using Hierarchical Channels

An expedited message’s capability to overtake other messages at the switch nodes can be implemented by employing *flooding* and *priority queues*. As soon as a copy of an expedited message reaches any of the *in\_queues*, it raises an interrupt and stores the identity of that *in\_queue* in a record at the *tail* of a list called *expedited-list*. Unlike other messages, the expedited message is placed at the *head* of the *in\_queue* along which it arrives rather than at the *tail*. In response to the interrupt, *sp* suspends the non-deterministic selection of *in\_queues* for a while (as explained in Section 4), and selects the record at the *head* of the expedited-list to get the identity of the *in\_queue* where a copy of the expedited message can be found. Then *sp* forwards copies of that message on their way to the destination by putting them at the *head* of all the *out\_queues* in the

*s-net* (rather than at the *tail*). It is possible that while one expedited message is being handled by *sp* other expedited messages arrive at the switch node. So, *sp* executes a loop: having processed the expedited message at the *head* of the expedited-list, it checks the list for the presence of other expedited messages and processes all of them before resuming the non-deterministic selection of *in\_queues*. When copies of an expedited message that has already been forwarded arrive along other *in\_queues*, they are ignored, i.e., no messages are propagated.

### Expedited Data Handling

```

type
  exp_record=record
    queue_number : integer;
    next          : ↑exp_record;
  end;

var
  exp_entry : ↑exp_record;
  head, tail : ↑exp_record;(initialized to NULL) /* head points to the head of the expedited-list */

```

### Code to handle arrival of expedited message into in-queue i

```

begin
  create new record of type exp_record and let exp_entry point to it;
  exp_entry↑.queue_number←i;
  add exp_entry to the tail of the expedited-list;
end;

```

### Code executed by sp to service the interrupt raised by expedited messages

```

begin
  repeat
    {x←head↑.queue_number;
    extract first expedited message from in_queue x;
    handle the expedited message just like other messages except that outgoing
    copies are placed at the head of all out_queues rather than at the tail;
    head←head↑.next;
    }
  until head = NULL;
end.

```

In this implementation expedited messages can overtake each other at an *sp* or within an *out\_queue*. But the level property is maintained with respect to other messages. The expedited messages correspond to the lowest level. So, ordinary messages cannot overtake them. By providing some extra logic at the switch nodes, especially in the queues, the aggressive overtaking of the expedited data transfer protocols is provided along with the flexibility of hierarchical channels.