ISSUES IN ORDERED MULTICAST PERFORMANCE: A SIMULATION STUDY

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

by
Michael Ḥusayn Kalántar
August 1995

ISSUES IN ORDERED MULTICAST PERFORMANCE: A SIMULATION STUDY

Michael Ḥusayn Kalántar, Ph.D.
Cornell University 1995

Process groups are an increasingly popular tool for programming distributed systems. Such groups consist of collections of processes that work together to provide reliability, fault tolerance, task distribution, or some other abstraction. A combination of group actions, reliable failure detection, and ordered message delivery are used to provide group functionality.

Causal ordering, one form of message ordering, is fundamental in process group systems. Its cost is therefore an important determinant of overall performance. In order to gain more insight into the behavior of causal order protocols, two simulations of process group systems were developed.

The first, a detailed simulation of all system processes, groups, and interconnection networks, was used to study small systems. Using this model, a number of parameters were identified as having the greatest impact on ordered multicast performance. These parameters were used as indices to a set of precomputed probability distribution tables used by the second simulation.

The second simulation focused only on key processes — packet originators and processes in multiple overlapping groups — and used precomputed probability tables to reduce simulation overhead. Simulations using this model showed that the delays imposed by the ordering protocols result in a tendency for packets to become "convoyed" together. Consequences are larger delays and greater system and network burstiness.

We speculate that this tendency for systems to become more bursty, or less uniform, is a general principle. That is, any system which delays actions on a "microscopic" level results in burstiness at a "macroscopic" level. The more times such delays occur, the greater the degree of burstiness. Such a principle would imply that systems with this characteristic are limited in scale by the size of the largest burst that can be handled. It also suggests that protocols that minimize the number and length of delays and which optimize the handling of bursts should be used. In the context of process group systems, burstiness can be limited by minimizing the number of groups through which a packet is filtered and by protocol designs which minimize the number of delays used to control out-of-order message arrival.

Biographical Sketch

Michael was born in Edmonton, Alberta, Canada in November of 1967. He attended the University of Alberta starting in 1985. He graduated with a BSc with Honors in Computing Science in 1989. After a summer of APL hacking in Toronto, he joined the graduate program at Cornell University. Despite studying at Cornell, he managed to find the time to return occasionally to Toronto where he had met a creature of heaven named Suzan. They married in the midst of a snowstorm in February of 1992. In December of 1992 Michael received a MSc, in April of 1994 he received Varqá, a son, and in August of 1995 he finally received a PhD. He and Suzan now go to Shandong University to teach.

سوان

Acknowledgements

First, and foremost, I would like to thank Ken for his support and guidance. He has been very patient, allowing me to work at my own pace and relatively independently. But he always perceived exactly the right times to offer words of encouragement and useful suggestions.

In addition to Ken's guidance and support, I thank all of the ISIS researchers who were always ready and willing to help me. They have helped me shape my thoughts through helpful discussions and have provided me with many insights into the "big picture." My classmates and various officemates have, over the years, provided me with much needed companionship.

Outside the realm of academia, I thank the Bahá'ís of Ithaca for their unconditional love and support. They have been my second family, my home away from home, and my constant source of energy. It is due to them that I have grown spiritually, as it is due to Cornell that I have grown intellectually.

A final word of thanks go to Suzan, my dearest wife. Without her constant attention and encouragement, I would have given up long ago. May she be granted eternal happiness and everlasting joy.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Process groups are an increasingly popular tool for programming distributed systems. Such groups consist of collections of processes that may be addressed as a single entity. The processes in a group work together to provide reliability, fault tolerance, task distribution, or some other abstraction. A combination of group actions, virtual synchrony, ordered message delivery, and reliable failure detection are used to provide group functionality. This dissertation, a technical result about the cost of message ordering, assumes a basic understanding of process group systems[1].

Causal ordering, one form of message ordering, is fundamental in process group systems. Its cost is therefore an important determinant of overall group performance. This dissertation focuses attention on determining the cost of protocols providing causally ordered message delivery within very large group based systems such as those found in information distribution or large process control environments. These costs have been experimentally determined in small systems. Further, the protocols have been experimentally tuned in a number of specific situations. Prior experiments fail, however, to provide much insight into their cost and behavior in large environments. In order to gain more insight into protocol behavior, two simulations of process group systems were developed.

The first, based on a detailed simulation of all the system processes, groups, and the interconnection networks, was used to study small systems. From this group size, sender frequency, load on sender, degree of stability, send window size, and

[1] One source about such systems is [Bv94] which provides an in depth study of ISIS, a group-based system.

the number of distinct destination groups were identified as the most important parameters affecting ordered multicast performance. These parameters were used as indices to a set of precomputed probability distribution tables used by the second simulation.

The second simulation focused only on key processes — packet originators, processes in multiple overlapping groups, and some packet sinks — and used the precomputed probability tables to reduce simulation overhead in large systems. Simulation of large systems showed that the delays imposed by the ordering protocols result in a tendency for packets that are forwarded through a number of groups to become clumped together. The consequences are larger delays and greater system and network burstiness.

The result that large systems tend towards burstiness was not expected, but is, perhaps, not surprising. Other complex systems demonstrate similar behavior. For example, database systems display a "convoy" phenomena in which transactions become grouped together [BGMP79]. We speculate that this tendency for systems to become more bursty, or less uniform, is a general principle. That is, any system which delays actions on a "microscopic" level results in burstiness at a "macroscopic" level. The more times such delays occur, the greater the degree of burstiness. Such a principle would imply that systems with this characteristic are limited in scale by the size of the largest burst that can be handled. It also suggests that protocols that minimize the number and length of delays and which optimize the handling of bursts should be used. In the context of group based systems, burstiness can be limited by minimizing the number of groups through which a packet is filtered, or by protocol designs which minimize the number of delays used to control out-of-order message arrival.

1.1 Group Based Computing

In computer systems where faults are likely, as is the case in many distributed systems, software must be designed to make progress in spite of failures; that is, programs must be *fault tolerant*. Two common ways in which fault tolerance is achieved are by replicating processes and by distributing a task among a group of cooperating processes. In the case of replication, each replica maintains a copy of the system state. A failure of one replica does not prevent the progress of the other replicas. In the case of cooperating processes, each process does a part of the whole task. The failure of some participants does not prevent the others from proceeding; though it may increase their workload because they may also need to do the work of the failed process.

However, replication and process cooperation are insufficient to guarantee fault tolerance. In both cases, it is necessary to be able to identify when a failure has occurred. Clients of a replicated service may need to know to which live replica to send requests. Replicas themselves may need to make joint decisions (for example, in database commit protocols, quorum decisions, and coordinator elections) which require knowing which replicas have failed. Live cooperating processes must determine when to do the work of failed processes. Further, they should probably avoid doing so unless the processes have really failed.

In addition to being able to identify failures, replicas or groups of cooperating processes may need to be able to reason about the order of actions at other replicas/members. Replicas are supposed to keep track of the system state. The consistency of different replicas may be a requirement for fault tolerance[2]. Maintaining consistency typically requires some form of event, or message, ordering. Cooperating processes need to know which members are doing which tasks so that all tasks are accomplished without task replication.

The minimum requirements for fault tolerant distributed systems are therefore the ability to structure systems into groups of cooperating processes (while in the preceding paragraphs a distinction was made between replication and cooperating process, replication is just a different form of process cooperation), to detect when process failures occur, and to order events within processes. A relatively new paradigm for implementing distributed systems with these characteristics is the *process group* paradigm. A number of systems and toolkits provide (some or all) basic primitives for group management, failure detection, and event ordering. Examples of such systems include V [CZ85], ISIS [BJ87b,Bir93], Psync [PBS89], Transis [ADKM92b,DM95], Horus [vHB94], Totem [AMMS+93], RMP [WMK94], Newtop [EMS95], and Relacs [BS95,BDGB95]. Such systems may also provide

[2]A consistency requirement is a function of the semantics of a particular system.

higher level tools for building reliable distributed software by easily interconnecting nondistributed programs. Such tools include those for replicating data, synchronizing distributed computations, automating recovery, and dynamically reconfiguring a system to accommodate changing workloads. However, these tools are built on top of the basic tools: group actions, fault detection, and event ordering.

Providing group actions, fault detection, and event ordering that have meaningful semantics will have a cost. In particular, this cost will be incurred even when the semantics provided are stronger than are actually required by a particular application. For this reason, it would seem that general process group mechanisms would have higher cost than other distributed programming paradigms such as RPC, client-server, or primary-backup. That this is the case is not clear: these other paradigms may require the programmer to provide the necessary failure detection and event ordering mechanisms (as in RPC) or they may be just special cases of a process group paradigm (as in client-server or primary-backup). The cost of such special mechanisms may be as high as those for a process group toolkit.

1.2 Multicast Ordering

As mentioned above, process group systems may provide a means for ordering events among the processes of a group. This can be done (in the case of deterministic algorithms) by ordering the messages sent to, and received by, process groups. Such messages, called *multicasts*, can be ordered in a number of ways: they may be unordered, FIFO ordered (for sender-receiver pairs), causally ordered, or totally ordered. A causal order is the order obtained by ensuring that every message is delivered only after all causally preceding messages have been delivered. A total order is an order obtained by ensuring that every message is delivered in the same order at all members of a group. The use of ordered multicast simplifies distributed systems programming. It is hypothesized that providing such ordering at the message level is the most efficient way to implement system ordering requirements. Because multicast messages are very common in a distributed system, their performance is fundamental to the overall performance of the system.

Guaranteeing causal or total order may require that processes delay messages and may require the exchange of additional messages. Examples of protocols which guarantee such orders can be found in [BJ87b], [PBS89], [KTHB89], [LLSG92], [BSS91], [Ste91], and [ADKM92b]. Because delays and additional messages may be required, it is expected that ordered multicast protocols are less efficient than unordered multicast protocols. There is, however, a vigorous debate surrounding both the utility and the cost of guaranteeing order; especially causal order [van93, CS93,Bir94,Coo94,van94,CS94,Shr94]. There are two aspects to the debate over cost. First, is the issue of large groups. In large groups there is a greater chance that the same multicast will arrive at widely different times at different destination processes. The consequence of this will be a greater probability of multicasts arriving out of order at some of the destinations. Second, is the issue of overlapping groups. In this case more information is needed to determine when messages arrive out of causal order. However, it is also desirable to limit the amount of ordering information piggybacked onto messages. A consequence of reducing the amount of piggybacked information is larger delays.

1.3 System Scale

In small systems the cost of the overhead for maintaining causal order is minimal because groups are small and because the amount of overlap is small. However, what happens in large systems will determine the usefulness of the process group paradigm for general distributed systems development. It may also establish limits on how large any distributed system requiring event ordering may be. This dissertation is a contribution towards understanding how limiting causal order is in a process group based distributed system.

In order to study the effect of scale on the performance of causal ordering within the context of the process group paradigm, we consider systems composed of a large number of relatively small groups. It may appear to be limiting to consider only small groups. However, this is not the restriction that it appears to be: a common method for constructing large groups is by using a hierarchy of small lightweight groups [Gla95].

Within the framework of a large number of relatively small groups, there are two approaches to the issue of scale. One approach is to consider many randomly sized groups with a high degree of overlap. Such a group structure might be found in a distributed object maintenance system such as [SF94]. In such a system, an object is represented by the group composed of the servers on which it is stored and the active clients at which copies are cached; objects may be stored on any number of servers and may be mapped into any number of address spaces. Because the number of machines (servers and clients) is much smaller than the number of objects, there is a high degree of group overlap. This overlap is highly random: it is a function of which machines are using which objects. Figure 1.1 shows a small example of such a system containing four objects. Consistently updating objects requires ordering messages across multiple groups.

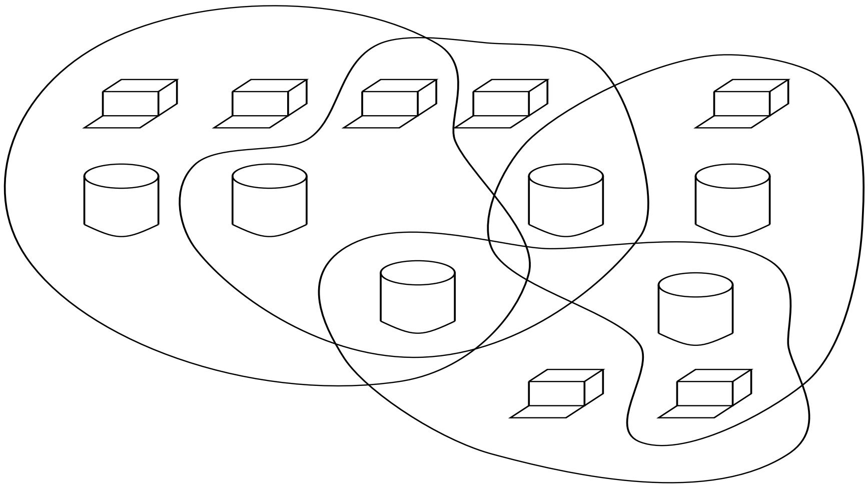


Figure 1.1: Random structuring of process groups used for distributed object store. Four objects are represented.

A second approach to investigating issues of scale is to consider a regular tiling of many similarly sized groups. Such a regular tiling would be found in large information dissemination systems such as the ISIS News system [Gla95] or in process control systems such as those found in factory floor settings or air traffic control applications. Examples of such structurings are presented in Figures 1.2 and 1.3. In the information dissemination example (Figure 1.2), a few of the processes in the system may be information sources. The remaining processes are information sinks which may (if in the intersection of groups) forward information to other information sinks. In the process control example (Figure 1.3), each group

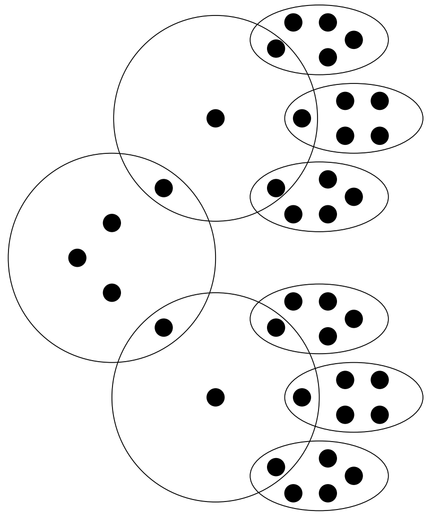


Figure 1.2: Hierarchical structuring of process groups used for information dissemination.

may represent a set of controllers for a particular task or sequence of tasks. The larger group represents an overall process controller.

1.4 Process Group Simulation

In order to study system performance, we present simulations of large systems that were undertaken using a custom built, event driven, network simulator. The simulator provides a language for describing a process group based system and its behavior, a template for the creation of simulation experiments that vary a number of parameters, a mechanism for the collection of statistics, and tools for analyzing the resulting data. In a simulation, data packets are created by specified processes and multicast to process groups enforcing a causal delivery order. Statistics are collected which measure latency, throughput, and message delays.

Two versions of the network simulator were developed. The first is based on a detailed simulation of each process in the system. This process-based simulation turned out to be useful only for small scale simulations because it was too

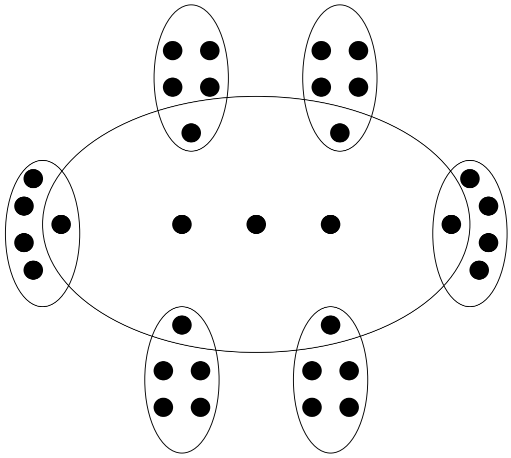


Figure 1.3: Typical group structure for process control applications.

computationally intensive. However, it served as the basis for the second, and less detailed, simulator. This second simulator is a group-based simulator: it eliminates the details of process-to-process communication. These details are replaced with probability distributions derived from the initial simulator.

1.5 Thesis Outline

The next chapter provides the framework in which the work was done. It formalizes the concepts of process group and causal order and introduces the language used to describe the protocols used to guarantee causal order. Finally, it outlines the particular protocols studied. Chapters 3 and 4 introduce the simulation models developed. Chapter 3 focuses on the process-based simulation model while Chapter 4 focuses on the group-based model. Each chapter introduces the model and discusses initial results obtained using it. Chapter 5 describes the results of the simulation of parts of a large, but regularly tiled, information dissemination system. This system is assumed to be a hierarchy (tree) of groups. Finally, Chapter 6 summarizes the work presented, discusses other approaches to studying issues of scale, and suggests directions for further research.

Chapter 2

Background Work

2.1 Introduction

This chapter discusses the framework in which the simulations were developed and experiments were run. It describes the concept of process group and gives examples of their common uses, introduces and defines ordered multicast, presents algorithms for guaranteeing causal order, and discusses issues relating to their performance. Finally, this chapter describes the Network Simulator [Hey90,Mar88], which was used as a starting point for the simulations developed here. Modifications made to the original simulator are briefly mentioned; they will be more fully described in Chapters 3 and 4.

2.2 Process Groups

A process group is an implementation of the logical notion of a set of cooperating processes. This collection of processes forms a communication endpoint; that is, messages can be addressed to the group as a whole (such messages are called *multicast* messages) in addition to individual members of the group.

Process groups are an abstraction of the dynamic nature of a distributed system. In a distributed system individual processes may fail, processes may be stopped for maintenance or software upgrades, networks may fail, failed processes may recover, new servers may become available, and new services themselves may become available. Process group implementations must handle such membership changes in a reasonable, and consistent, way. A number of membership protocols have been developed to do this. Examples include [ADKM92a,DMS94,MMSA91, RB91,Ric93,MPS91].

In this work, groups are assumed to have fixed membership; that is, the systems considered are static in nature. This assumption is a simplification of the general problem — group membership changes affect ordered multicast performance because multicast messages may be ordered with respect to group view changes and because may result in additional messages. As a first step, such effects are ignored.

Experience with one process group toolkit, ISIS [BC90], suggests that most uses of groups can be divided into four categories: peer groups, diffusion groups, client-server groups, and hierarchical groups. These four uses are depicted in Figure 2.1 [BSS91]. A fifth trend is to use many randomly ordered groups [SF94] in object oriented systems. In this work we consider large systems composed of hierarchically structured groups and of peer groups. We do not consider systems composed of randomly ordered groups because their irregularity makes them more difficult to study.

2.3 Ordered Multicast

A *multicast* message is a message sent to a set of recipient processes. Such a message is a *point-to-point* message if there is just one recipient and is a *broadcast* message if the set of recipients is all the processes[1]. In the context of process groups, a multicast is a message sent to a process group.

Multicast messages may be implemented in a number of ways. They may be implemented by a set of point-to-point messages, by using a hardware broadcast facility, by using a hardware multicast facility, or by a combination of these techniques. In the simulation studies done here, both hardware and software multicast implementations were considered.

A point-to-point implementation is a software solution in which each multicast

[1]The term broadcast is most commonly used in the context of a "broadcast" network, a network in which every message can be detected (and read) by every host attached to the network. A broadcast is a single message sent to, and received by, all hosts attached to the network. The term broadcast is, however, frequently used to mean a message sent to some set of processes. In this dissertation, we use multicast for this.

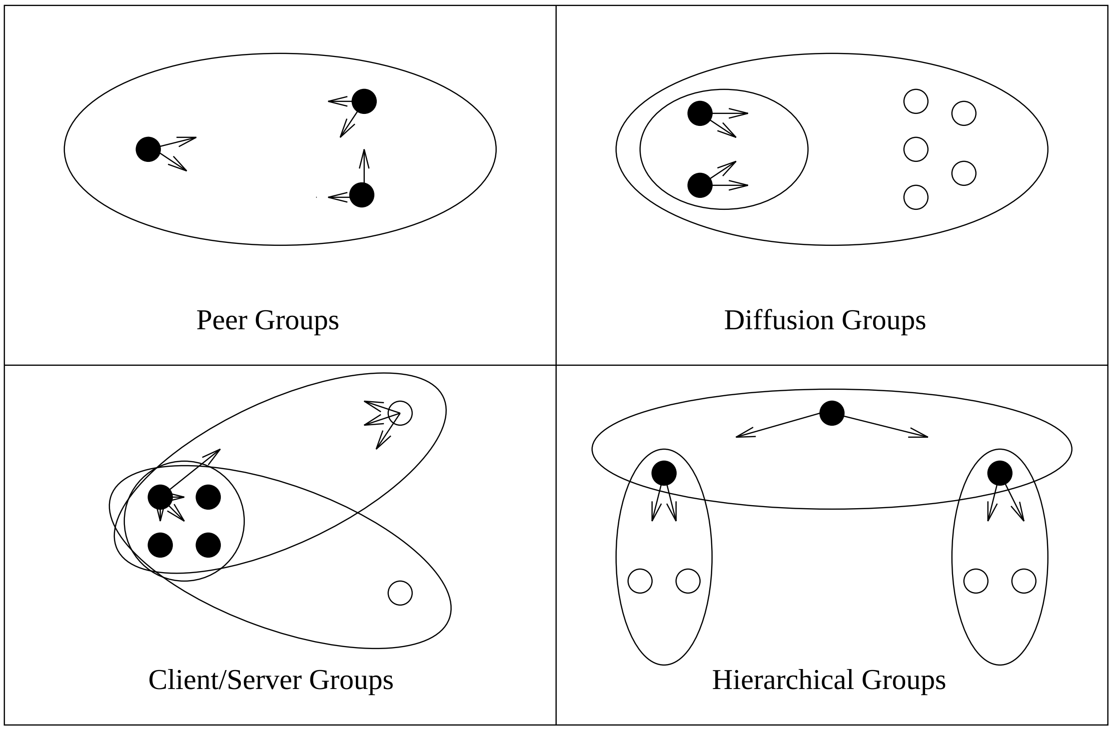


Figure 2.1: Common uses of groups can be divided into four categories. Processes in *peer groups* cooperate as equals in a closely coordinated distributed action. In a *client/server group* a peer groups acts as a server on behalf of many clients. A *diffusion group* is a special case of a client/server group in which servers multicast messages to all the clients (and other servers). Finally, *hierarchical groups* are tree structured set of groups used in systems with large server groups.

is sent point-to-point to each destination process. A software implementation has the disadvantage that it requires multiple network accesses for each multicast message. The network is then more likely to be congested. Software multicasts have, for example, been used in the ISIS toolkit until recently.

The broadcast approach takes advantage of broadcast networks such as Ethernet [MB76,IEE85a], Token Ring [IEE85b], or FDDI [ISO89]. In such networks a special address is used to send messages to every machine on the network. Each machine forwards received messages to destination processes resident on that machine. To multicast a message to a subset of the processes on a local network, each machine must maintain a list of which local processes are in which destination groups. Messages are interpreted by the machine to identify the address of the destination group. In this approach machines must handle (in software) many messages not destined for any process executing on it. This leads to high message loss rates if the number of messages sent (by all senders) is high[2]. This approach, as described, is limited to a single LAN. It can, however, be extended to multiple LANs by using bridges to forward broadcasts. An example of the use of such protocols is in Transis [ADKM92b].

Many broadcast networks also provide a multicast facility in which network addresses can be assigned to a group of more than one machine. The network hardware can then rapidly determine if a message is intended for a process on that machine. This method has the advantage of not being a software solution in local environments. It can be extended to non-local environments through software such as IP multicast [Dee89,DC90]. The ISIS toolkit now uses a combination of IP multicast and software multicast [BC94].

Transport level point-to-point message passing systems typically provide reliability and message ordering. That is, they typically guarantee that messages are received in the same order that they were sent. Corresponding properties for multicast transport systems are, respectively, atomicity and message ordering. Atomicity is the property that if some non-faulty destination process receives a message, then every non-faulty destination process does so. Two message ordering properties are the most natural candidates. One, *causal ordering*, is the extension of the FIFO ordering found in point-to-point systems to a partial ordering according to the "happens before" relation [Lam78]. The other, *total ordering*, is the ordering of all messages. These orderings are discussed in more detail in the following sections.

[2]Large numbers of broadcasts (by many senders) results in a phenomena called a "broadcast storm." Many back-to-back messages (from different senders) may be delivered at all machines overwhelming the software and perhaps the hardware. The result is a very high message loss rate by all machines. This problem does not occur with a single sender because a single sender is typically required to insert a gap between successive sends. This gap is large enough to prevent this problem.

2.3.1 Causally Ordered Multicast

Causal order can be thought of as a generalization of FIFO order [van93]. In a two-process system, a FIFO ordering requirement guarantees that a message A sent prior to a message B (by the same sender) is received before B is received. A causal ordering requirement guarantees that the same thing occurs in a system with more than two processes. However, messages A and B are multicast messages which may originate at different processes.

Multicast messages are said to be causally ordered if their delivery is ordered according to the *potential causality relation* ($\rightarrow$). This relation, Lamport's "happens before" relation [Lam78], is defined as the transitive closure of:

1. If $\exists p : e \xrightarrow{p} e'$, then $e \rightarrow e'$

2. $\forall m : send(m) \rightarrow deliver(m)$

where e and e' are events, m is a message and $e \xrightarrow{p} e'$ denotes event e precedes event e' at process p.

The *causal delivery ordering rule* states that if two messages m and m' are sent where m' causally depends on m, then m must be delivered everywhere before m'. This is:

$$send(m) \rightarrow send(m') \Rightarrow \forall p \in dests(m) \cap dests(m') : deliver(m) \xrightarrow{p} deliver(m')$$

Note that this rule does not constrain the delivery order of two concurrent multicast messages — concurrent messages may be delivered in different orders at different destinations.

2.3.2 Totally Ordered Multicast

A causal order is a partial order. Consequently, concurrent multicasts may be legally received in different orders at different destinations. A total order guarantees that this does not occur — even concurrent multicasts are received in the same order at all (overlapping) destinations. Formally, the *total delivery ordering rule* is,

$$\exists q : deliver(m) \xrightarrow{q} deliver(m') \Rightarrow \forall p \in dests(m) \cap dests(m') : deliver(m) \xrightarrow{p} deliver(m')$$

2.3.3 Measuring Multicast Costs

This dissertation looks only at the performance of causally ordered multicasts, particularly in situations where process groups overlap. No attempt is made to argue that causal order is useful; this task is left to others (see for example [van93, Bir94,Coo94,van94]).

One argument against using causally ordered multicast [CS93] is that their cost is too high for large systems. Despite this debate, few studies have been carried out to measure the actual cost of guaranteeing causal order in large systems. A number of performance studies of ordered multicast protocols have been done [KTHB89, ADKM92b,PBS89,BSS91,BC94,WMK94,AMMS+93,vHB94]. All of these studies consider the behavior of multicast in the context of a single group. Some consider only the cost of guaranteeing total order [KTHB89,AMMS+93,WMK94,vHB94] while three present measurements of a causal order protocol [PBS89,BSS91,BC94]. The use of hardware multicast is considered by [ADKM92b,BC94,WMK94,vHB94].

In addition to the studies measuring ordered multicast performance, there have been a number of analytic cost analyses of ordered multicast [BSS91,LLSG92, CS93,May92,MMS94]. Of these, [May92] and [MMS94] attempt to model systems to determine the expected throughput of totally ordered multicast. [BSS91] and [LLSG92] count the expected number of messages needed for each ordering strategy but do not consider the expected length of delays. Finally, [CS93] considers what happens assuming very high throughput. In many ways these analytic studies are a poor way to measure performance. This is for two reasons. First, implementations of protocols may take advantage of, or benefit from, other system characteristics. For example, group systems, in addition to multicast protocols, typically send periodic "I am alive" messages. These periodic messages can be taken advantage of (by piggybacking information on them) to reduce the cost of additional messages. Hence the number of additional messages created by a causal order protocol may not be as high as one would expect from a description of the protocol. Further, many systems "batch" small messages into larger messages, effectively increasing message throughput, reducing ordering problems, and minimizing the transmission of control information. Secondly, asymptotic cost analyses, such as [CS93], tend to consider what happens in the limit. However, individual components of a larger system have practical limits. These limits may both limit scalability and may place an upper bound on the cost of a particular protocol. Indeed, the limiting cost of a protocol may actually enable the system to scale better. For example, a practical limit on the number of messages a process can handle places a limit on the amount of delay that may be incurred by a causal order protocol. However, this may not limit the size of a system built using process groups (rather, the size of the groups may be the limiting factor).

The purpose of this dissertation is to better gauge the "real" cost of causal order in process group systems. The use of simulation enables us to distinguish between different components of a group based system. In this case, it is used to isolate the costs of causal order protocols from the costs of fault detection and group membership protocols. Finally, we look at the cost of maintaining causal order in large systems where there are many overlapping groups. The effect of the delays caused by this overlap is studied.

2.4 Causal Order Protocols

Causal order protocols provide guarantees of causally ordered delivery to process group members. These protocols can be implemented at either the transport or application layer. At the user level they are typically implemented by library routines compiled into an application.

There are three different approaches to implementing causal order protocols. All rely on the basic idea that a process sending a message m must also send information about any messages causally preceding m. The first approach [BJ87b] solves this problem by piggybacking all causally preceding messages on message m. When m is received, it can be delivered immediately (without delay) because it contains all causally preceding messages. However, it means that messages are very large. The second approach, used in ISIS [BSS91], Horus[3] [vHB94], and in the gossip architecture [LLSG92], involves timestamping m with *vector timestamps*. These vector timestamps indicate how many messages, and from which senders, causally precede m. Upon receipt of m a process can determine how many messages

[3]Horus provides a general framework in which protocols can be layered in "stacks." Any of many causal order protocols can be used in different applications built using Horus [vB95].

it must wait to deliver before delivering m. This method is further described below. The final approach, used in Psync [PBS89] and Transis [ADKM92b], involves maintaining a shared graph of causal relationships. The creation of m involves adding information to this graph. The added information (known initially only to the sender) is propagated to other processes by piggybacking it on m and on messages which causally succeed m. In essence, this information is a compressed vector timestamp (see Section 2.4.3). That is, this approach is an optimization of the second approach.

Because the first approach is clearly inefficient and the third is essentially a refinement of the second, we use protocols modeled after the second approach in our simulations. These protocols are presented in two parts. First the most basic (optimistic) multigroup protocol, CBCAST, is presented. A number of optimizations to this protocol are then described. Of these, the *conservative protocol* and the *k-stable conservative protocol* (defined in Section 2.4.3) are of particular interest because they are the ones studied here. Before discussing these protocols, however, a number of terms and concepts are defined. They are presented because they are necessary to understand the details of the protocols and will be used throughout the rest of the dissertation.

2.4.1 Some Definitions

A number of the terms used to describe and discuss causal order protocols are presented. The relationship between the terms may not be immediately clear. It will become more clear when the protocols are discussed in Sections 2.4.2 and 2.4.3.

Vector Clocks

Vector clocks, an extension of Lamport clocks [Lam78], are a form of logical clock that precisely model causal event ordering [CB91]. A vector clock is a vector of logical clocks. A process p_i's j^{th} logical clock represents the number of events of process p_j whose causal effect has been observed by process p_i. If we wish to order messages, the events of interest are send events and deliver events[4]. Process p_i's

[4] A deliver event is when a message is delivered to its destination process. This is distinct from a receive event when it is received at its destination machine, or by the ordering software. It may be delayed after reception (and before delivery) in order to guarantee ordered delivery.

vector clock's j^{th} entry would therefore represent the number of messages sent by process p_j whose causal effect has been observed by process p_i. In particular, p_i's i^{th} entry would represent the number of messages sent by p_i.

Formally, given a set of processes $P = p_1, ..., p_n$. A vector clock for a process p_i, $VT(p_i)$, is a length n vector ($n = |P|$) indexed by process-id that behaves according to the protocol in Figure 2.2.

1. $VT(p_i)$ is initialized to zeros.
2. for each $send(m)$ event at p_i, $VT(p_i)[i]$ is incremented by 1.
3. message m is timestamped with $VT(p_i)$ (after it is incremented).
4. Upon delivery at p_j of m from p_i, p_j modifies its vector time by:

$$\forall k \in 1 \ldots n : VT(p_j)[k] = max(VT(p_j)[k], VT(m)[k])$$

Figure 2.2: Vector Clock Protocol

Vector clocks, which can be implemented as simple integer arrays, are a particularly useful tool for identifying causal relationships because they can be easily compared according to the rules found in Figure 2.3. Intuitively, a vector clock

1. $VT_1 \leq VT_2$ iff $\forall i : VT_1[i] \leq VT_2[i]$
2. $VT_1 < VT_2$ if $VT_1 \leq VT_2$ and $\exists i : VT_1[i] < VT_2[i]$

Figure 2.3: Rules for comparing vector clock values.

VT_1 is less than another one, VT_2, when VT_2 records the causal effect of more messages (including the same messages) than does VT_1. VT_1 is less than or equal to VT_2 when VT_2 records the causal effect of at least as many messages as does VT_1. Figure 2.4 details how this property of vector clocks is used in causal ordering protocols.

Multicast Stability

A multicast is said to be *fully stable* when it is known (by its sender) to have been received, though not necessarily delivered, by all its non-faulty destinations [BSS91]. A multicast is said to be *k-stable* [BSS91,Ste91] if it is known that the multicast has been received, though not necessarily delivered, at at least k destinations.

If we assume that the transport protocol provides atomic message delivery, all multicasts eventually become stable. The sender of a multicast can, if it so desires, then propagate this information to the recipients of its multicasts. This can be economically done by propagating a *stability sequence number*, $stable_i$, which is the largest integer such that all multicasts sent by p_i with $VT(p_i)[i] \leq n$ are stable. The value $stable_i$ can be propagated via piggybacking or, if p_i is not sending any messages when $stable_i$ changes, by a special stability control message containing only $stable_i$.

Active Groups

A group g is *active for process* p if p is engaged in an unstable multicast to g. That is, if:

1. p is the initiator of a multicast to g that is not stable, or
2. p has received an unstable multicast sent to g.

Note that activity is a *local property* if senders propagate stability information. That is, a process $p \in g$ can compute whether or not g is active for p by examining local state.

2.4.2 Basic CBCAST Protocol

CBCAST algorithms can be of two basic flavors, *conservative* and *optimistic*. In the conservative flavor, it is assumed that packets will arrive out of order. Packets are therefore delayed prior to send until there is no chance that this can occur. An optimistic protocol assumes that packets will arrive in order and therefore sends them as soon as they are created. Of the two, the conservative protocol [BSS91] is studied more extensively here. It is a variant of the more basic optimistic protocol presented here and will be discussed in Section 2.4.3. Both protocols apply to multiple, overlapping groups. In this work, they do not take into account group membership changes. The protocols can, however, be modified to do so.

Briefly, each process maintains a vector clock for each group in the system. When a process sends a multicast message, it timestamps the message with all of its vector clocks. A comparison of the timestamp on a received message and the recipient's local vector clocks determines when the message can be delivered. By comparing vector clocks, the recipient can ensure that a message is delivered only when it has seen the causal effects of the same messages as the sender (the timestamp shows this). The details of the algorithm are in Figure 2.4.

1. Each process p_i maintains a vector clock for *every* group in the system. Denote p_i's clock for group g by $VT_g(p_i)$. These clocks are all initialized to zero at the beginning.
2. Before sending a multicast m to a group g, process p_i increments $VT_g(p_i)[i]$ $(\forall g : p_i \in g)$ and timestamps m with *all* of its vector clocks.
3. When a process $p_j \in g_a$ receives m from $p_i (\neq p_j)$ it delays delivery of m until:
   (a) $VT_{g_a}(m)[i] = VT_{g_a}(p_j)[i] + 1$,
   (b) $\forall k : (p_k \in g_a \wedge k \neq i) : VT_{g_a}(m)[k] \leq VT_{g_a}(p_j)[k]$, and
   (c) $\forall g : (p_j \in g) : VT_g(m) \leq VT_g(p_j)$
4. When a message m is delivered at p_j, $VT_g(p_j)$ is updated (for all g) according the vector clock protocol (Figure 2.2).

Figure 2.4: Optimistic CBCAST protocol.

This optimistic protocol is inefficient for two reasons. First, a vector clock for each group in the system must be maintained by each process and is attached to each message in the system. For those groups which never communicate with each other, there is no need to maintain or propagate all this information. An analysis of the communication structure [BSS91] or the use of causal domains [BSS91,BCG91], can limit this inefficiency. Second, all the vector clocks are attached to messages even when many of them may not have changed. This inefficiency can be reduced by transmitting only those entries that change. A consequence of this policy is that destination processes must maintain information about the vector clocks of those sending to it. The next section considers these, and other, modifications to this basic protocol. Most of these modifications are complex and so are not used in practice.

2.4.3 CBCAST Variants

The optimistic CBCAST protocol is inefficient because too many vector clocks must be maintained and sent with each message. Five variants of the optimistic protocol which reduce the number of vector clocks maintained and sent with each message are presented here. These are: basic vector timestamp compression, extended vector timestamp compression, the conservative protocol, the k-stable conservative protocol, and the technique of multicast epochs. Except for the conservative protocol, which is of particular interest in this work, the optimizations are only briefly summarized. Further details about them may be found in [BSS91,Ste91].

Basic Vector Timestamp Compression

Vector clocks can be compressed by eliminating from messages those vector clock entries which do not change. That is, when p_i sends multicast m, only those vector clocks whose fields have changed, and only those fields, are included in the timestamp. Further, if p_i has not received anything between two sends (recall that timestamps change only on send and on delivery), no timestamp is sent. An assumption of FIFO channels ensures correct order.

Extended Vector Timestamp Compression

Extended vector timestamp compression extends basic vector timestamp compression by reducing the number of vector clocks that form the timestamp in the first place. Timestamps of inactive groups are omitted from multicast messages. The information supplied by including the vector clocks of inactive groups records the fact that there may be a causally preceding stable message (because the group is inactive). However, since the messages are stable, any overlapping destinations are already aware of the messages — they must have already received the messages.

Conservative Protocol

The conservative protocol extends the extended vector timestamp compression technique by eliminating the need to transmit any vector clocks except the sender's vector clock for the destination group. It does so by restricting the sending of new multicasts if it is possible that a causally preceding multicast may be delivered first (at some destination). The *conservative multicast rule* [BSS91] formally states that a process p may multicast to g if and only if g is the only active group for p or p has no active groups. During the delay before sending, no received messages are delivered. This rule ensures that all groups eventually become inactive for p, allowing p to send its multicast.

By preventing multicast sends in this manner, the conservative protocol can eliminate most vector clocks from its messages. The protocol ensures that no unstable causally preceding message from another group can arrive at common destinations once a message is sent. It is information about such messages that is omitted from the timestamp.

In essence, the conservative protocol delays a send if two successively causally dependent messages are sent to two different groups. Locality of process-to-process messages [Mog92] suggests that, with high probability, a multicast message sent will be to the same destination as the message immediately preceding it, limiting the length of this delay.

k-stable Conservative Protocol

The conservative protocol delays multicasts until full stability is achieved. This can be optimized by delaying only as long as it takes to achieve k-stability. Using k-stability is, in essence, making an assumption about the likelihood of more than k concurrent failures among the members of a single group during a multicast. If it can be assumed that fewer than k failures occur then fault tolerance is not sacrificed by waiting only for k-stability. When a k-stable multicast is used, we will also say that the multicast has stability of degree k.

Multicast Epoch

In the case where few messages are expected to be out of causal order they may be timestamped with an *epoch number* instead of with a vector timestamp. A sender assigns a higher epoch number to messages when there is a potential causality conflict (this can be determined as in the conservative protocol by looking at the active groups of the sender). Thus, the delivery condition is that all messages with lower epoch numbers must be delivered before those with higher epoch numbers (this protocol also assumes FIFO channels). This is achieved by use of a group flush[5], initiated by a recipient who has received a message with a higher epoch number, to ensure all lower epoch numbered messages are delivered first. The multicast epoch protocol eliminates the direct delay at the sender and replaces it with an indirect one that results from a flush protocol.

2.5 Network Simulation

A process group simulator implementing causally ordered multicast was built using the discrete event-based *Network Simulator* [Hey90,Mar88] as a base. This section describes the original Network Simulator and summarizes the modifications that were made to predefined parts of it to provide process group and ordered multicast semantics. These modifications are further described in Chapters 3 and 4.

2.5.1 The Network Simulator

The Network Simulator is an event-based simulator for simulating events of any system that can be modeled by a network of components that send messages to each other. The user interface to the simulator [Mar88] provides an X Windows interface which displays the topology of the network and the parameters of its operation, allows the user to modify the network and control the simulation, and to save and load the network configuration. The core of the simulator consists of an event manager and various tools (generic lists, queues, hash tables, I/O

[5]A group flush is a mechanism by which all "active" messages within a group can be forced to become stable before any new messages are initiated by the group members. See [BSS91] for details.

routines, etc.) that can be used to define components, the basic building blocks of a simulation experiment.

A *component* is a basic entity in a simulation model. For example, processes, process groups, machines, and networks are components. Each component is implemented as a data structure and a set of action routines. An instance of a component's data structure contains the state information about that instance. Action routines correspond to the actions to handle each event that occurs at a component. In general, an action will change the state of at least one instance of the component. Components interact by scheduling events (calling action routines) for each other.

Components of the original Network Simulator that were used or modified for the models we developed are now briefly described. A brief description of the modifications made to them is presented. More details are provided in Chapters 3 and 4.

The components in the original simulator are:

psource: A user level process that generates packets using a Poisson source. Such processes send packets in a point-to-point manner to packet sinks.

sink: A user level process that accepts packets but does nothing else.

host: A machine's communication interface to the network. The basis of this interface are input and output queues to the network device.

etlink: An Ethernet and its interface to hosts. An Ethernet [IEE85a] is a non-slotted[6] 1-persistent[7] carrier sense, multiple access with collision detection (CSMA/CD) network. Collisions result in a random backoff before active senders attempt to retransmit. `etlink` is a simulation of a non-slotted, 1-persistent CSMA/CD network. However, upon collision, hosts do not retry some random backoff. Instead, the network queues colliding packets and retransmits them as soon as the network is free[8].

[6] A *slotted* network is one in which there is a master slot-clock and transmissions may begin only at slot boundaries. A *non-slotted* network is one in which transmission starts may be separated by less than the slot-time.

[7] A network is *1-persistent* if a host that becomes ready to transmit when the channel is busy will, with probability 1, transmit as soon as the channel is free.

[8] Such a capability does not really exist in an Ethernet. The consequence of this assumption

We can think of the `psource` and `sink` components as user application processes, the `host` components as the machines on which these processes execute, and the `etlink` component as the network that connects them together. This set of components was modified to provide ordered multicast semantics.

2.5.2 Summary of Modifications

Process group simulation was achieved by defining three new components, a `process` component, a `group` component and an `atm` component. Each is now described.

A `process` component combines the `psource` and `sink` components. That is, a `process` component may send and receive packets. Packet sending was, however, modified to allow fixed intervals between sends and normally distributed burst sizes in addition to the Poisson sending frequency. Packets can also be sent in response to other events such as sending, receiving, or delivering another packet. Further, a two-tiered sliding window flow control algorithm, similar to that used in ISIS [BC94], was built into the component. This algorithm attempts to piggyback acknowledgements of both message reception and message delivery on outgoing messages. Finally, the k-stable conservative causal ordering protocol (for all k) was implemented. This implementation includes maintenance and propagation of vector timestamps and stability information. Whenever possible this information is piggybacked on outgoing messages.

The `group` component defines statically sized process groups. Messages addressed to a group are sent to all members of the group, either by hardware or software multicast. In the case of hardware multicast, the `group` component is logically placed between the network and the recipient processes. The component is logically placed between each sending process and the network when software multicast is used. Finally, a process may be a member of many groups. It can send messages to each group according to different rules.

Finally, an `atm` component was defined to simulate the behavior of an high speed network such as an ATM LAN [dP93a,HHS94]. The model used is the same as for

will be to make the model more efficient than a real Ethernet. Throughput will be higher and delays smaller. However, this will serve to further minimize the number of packets delayed at reception and to illustrate the magnitude of the measured delays caused by the ordering protocols.

the `etlink` component except that the network bandwidth is assumed to be higher (155 Mbits/sec vs. 10 Mbits/sec). This approach approximates an ATM switch when each host's communication paths (virtual circuits) share a common link. Clearly, this assumption is unrealistic for every host and every path. Therefore, in using this assumption, we insist that only a small number of hosts are actually attached to a particular `atm` component and we allow each host to be connected to multiple components; ie, to have multiple communication paths.

2.6 Conclusion

This chapter provided the background to the simulation work to be presented in subsequent chapters. The concept of process groups was presented and the reasons they may be useful in distributed systems were discussed. A related concept, ordered multicast protocols, was then discussed. In particular, causal multicast was defined and a number of protocols for guaranteeing causal order were presented. Lastly, the foundation of the simulation itself, the Network Simulator, was presented and the basic modifications made to it were described.

This background makes it possible to now look in detail at the two models of simulation implemented: a detailed simulation of all the processes in a system (Chapter 3) and a less detailed simulation of a subset of them (Chapter 4).

Chapter 3

Process-Based Simulation Model

In order to investigate the performance of causally ordered multicast in a process group environment a simulation of a group system was developed. This simulation was developed out of a desire to address two fundamental questions regarding causally ordered multicast. The first is the question of how frequently do multicasts arrive out of causal order. If, in percentage terms, it is difficult to make packets arrive out of causal order, then incurring some cost for those that do may not greatly impair performance. The second is the question of measuring the cost of those packets that actually arrive out of order. The cost metrics used here are the lengths of the delays incurred by the packets, the lengths of the added vector timestamps, and the number of extra messages needed to guarantee order. These questions are addressed in the context of a small number of small process groups in a single LAN environment. These restrictions are due both to a simplification and to limitations of the simulation model developed here. Chapter 4 presents a refined model for simulating large systems.

This chapter presents the simulation model, discusses the experiments that were developed to address the above questions, and presents the results of those experiments. Finally, limitations of the simulation model are identified laying the groundwork for a new model (discussed in the next chapter).

3.1 Simulation Model

We can view a system of process groups as a directed multigraph $CS = (P, E)$ where the nodes, P, of the graph correspond to processes. There is an edge $< p_1, p_2 >_g$ in E for each group g containing both processes p_1 and p_2 and in which process p_1 multicasts messages to g (and therefore to p_2). For example, the group structure in Figure 3.1(a) would be represented by the graph in Figure 3.1(b). The process-based model developed here implements this view of a system.

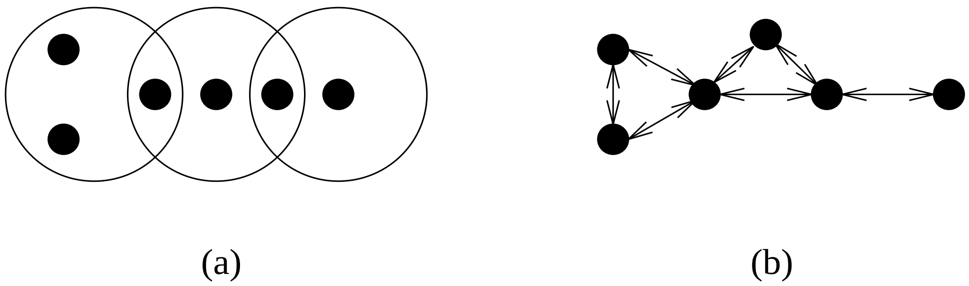


Figure 3.1: Example of group structure and its corresponding multigraph representation.

Each process, host machine on which the processes reside, and the network are simulated. Further, each packet in the system is traced from creation, through these components, until it has been received, delivered, and become stable.

Our model was implemented in C using the Network Simulator [Hey90,Mar88] as a basis. The Network Simulator provided the basic event handling routines that would have been provided in a conventional simulation package. Furthermore, it provided various generic tools, such as lists, queues, hash tables, and efficient memory management, that were used to construct the various components of our model.

The event graph [Sch92] in Figure 3.2 depicts our simulation model. This graph is elaborated upon in Figures 3.3 through 3.6. In brief, an event graph illustrates the simulation events (the nodes), the changes of state caused by the events (not shown in detail in Figure 3.2; instead a brief description of what each event means is provided in Figure 3.3), and the relationship between events (the directed edges or arrows). An arrow (dashed arrow) from node A to node B indicates that event

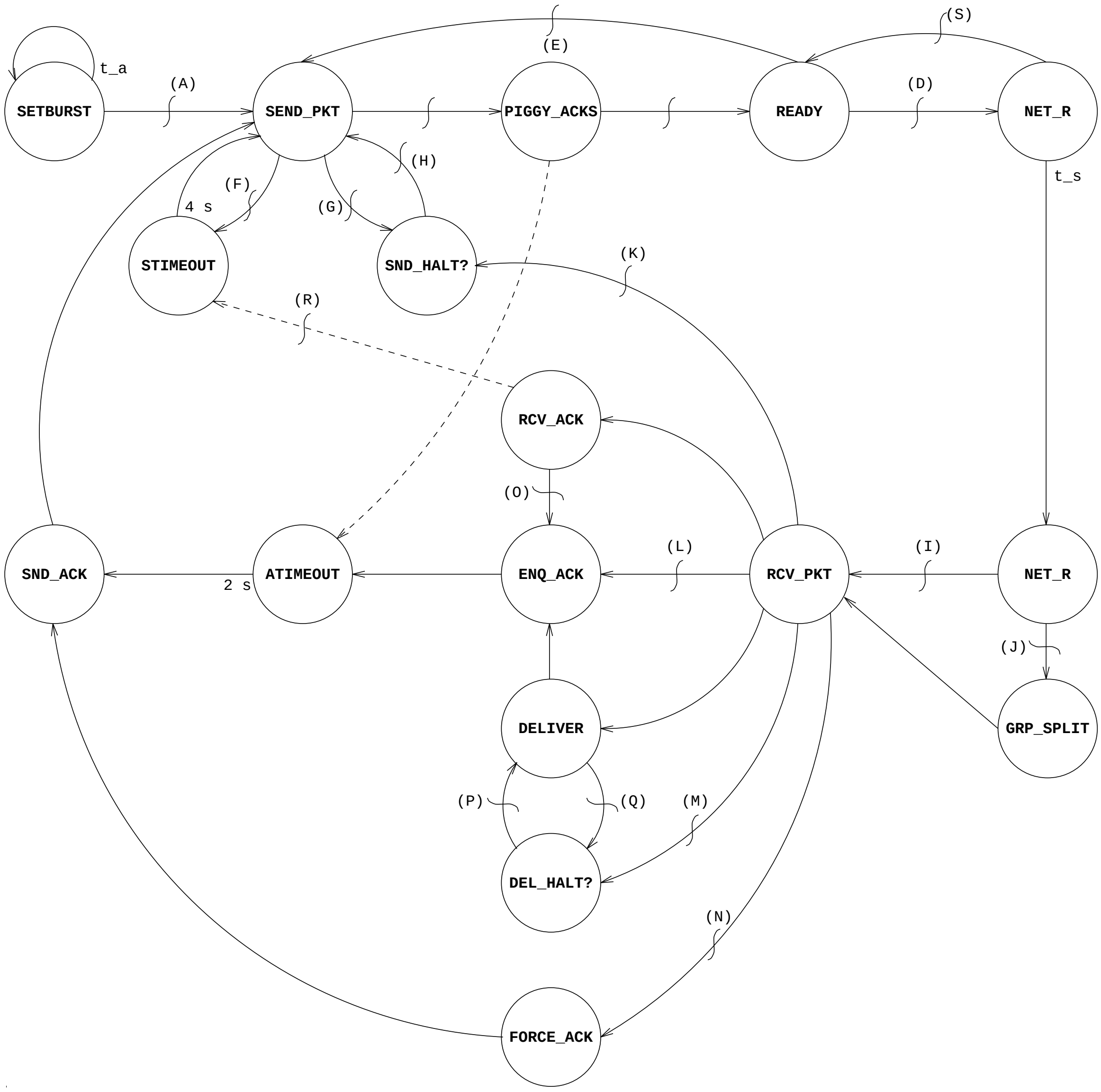


Figure 3.2: Verbal event graph for multicast simulation. The nodes represent simulation events, solid arrows the scheduling of new events, and dashed arrows the cancellation of events. Edges are annotated with scheduling information: timers and/or conditions under which events are scheduled/cancelled. Further explanations of the graph can be found in Figures 3.3, 3.4, 3.5, and 3.6.

SETBURST	begin a burst of data sending
SEND_PKT	create and prepare to send a packet
PIGGY_ACKS	piggyback any acknowledgements and control messages
READY	host machine forwards packet to network
NET_S	network begins sending packet
NET_R	network receives packet (from itself)
GRP_SPLIT	split message into copies for individual group members
RCV_PKT	handle a received data packet
RCV_ACK	handle a received acknowledgement
ENQ_ACK	note need to send an acknowledgement
DELIVER	deliver data packet
DEL_HALT	delay delivery awaiting ordering condition
FORCE_ACK	send acknowledgements immediately (receive window full)
ATIMEOUT	timeout hoping for acknowledgement to be piggybacked on another packet
SND_ACK	create acknowledgement message to be sent
SND_HALT	delay send while awaiting stability
STIMEOUT	timeout while awaiting acknowledgement of sent packet

Figure 3.3: Explanation of the events in the verbal event graph in Figure 3.2.

t_a	time between bursts (may be exponentially distributed)
t_s	time to transmit (computed)
4 s	retransmission timeout period
2 s	timeout before acknowledgement sent

Figure 3.4: Explanation of timing abbreviations used in the verbal event graph of Figure 3.2. These numbers represent the delay between the scheduling of an event and the time it occurs.

A, when it occurs, may cause event B to occur (may cancel a scheduled event B). The scheduling of event B may depend on the state of the system and may be delayed. This is shown in the event graph by annotating arrows with conditions and timing information. The annotations for our model are explained in Figures 3.5 and 3.4.

In implementing our model, some events in Figure 3.2 were combined into single events. Further, they were grouped according to the type of component they were

```
(A)  ≡  (number sent < send window)
(D)  ≡  (network free)
(S)  ≡  (hostQ)
(E)  ≡  (sendQ & !hostQ)
(F)  ≡  (type(pkt) in {data, dack_pkt})
(G)  ≡  (source ~active for dest group)
(H)  ≡  (source active for dest group)
(I)  ≡  (type(dest) == PROCESS)
(J)  ≡  (type(dest) == GROUP)
(K)  ≡  (halt)
(L)  ≡  (packet is new)
(M)  ≡  (halt)
(N)  ≡  (num unacknowledged > .8*receive window)
(O)  ≡  (type(ack) == dack_data)
(P)  ≡  (!halt)
(Q)  ≡  (halt)
(R)  ≡  (type(ack) == (dack_data || rack_data))
```

Figure 3.5: Pseudocode explanation of conditions in the verbal event graph of Figure 3.2. These codes define the conditions under which events are scheduled or cancelled. The lack of a condition on an edge implies that the condition is always true.

related to. Component types include process, group, host machine, and network. Each component is now described in more detail.

The key component of the model is the `process` component. A process is both a packet source and a packet sink. Packets can be generated in either fixed or random sized bursts at either fixed or random (Poisson distribution) intervals. They can also be generated in response to other events such as send, receive, or deliver events. Network contention is controlled by a two-tiered window based flow control algorithm. This algorithm is based on the one used in ISIS [BC94]. The current implementation uses fixed sized send and receive windows and fixed timeouts for (positive) acknowledgement piggybacking. Lastly, processes implement the k-stable conservative causal order protocol (which is the fully conservative protocol when k is the size of the group).

The specific functionality of a process is controlled by a large set of parameters.

1. The verbal event graph does not contain information about the state changes caused by each event. In general, these are bookkeeping activities; recording information about packets and acknowledgements in the system.

2. Each of the events really changes the system state at a few processes. Thus each event should be parameterized with a process identifier. This has been left out for simplicity.

3. For simplicity, information passed as parameters is not shown. Typical parameter information would refer to a data packet or acknowledgement that is passing through the system.

4. At each node, if the conditions on the outedge are not matched, the information passed (packets, acknowledgements) will typically be enqueued and sent later (more than one thing can be dequeued per event). These queues have not been made explicit on the event graph. For details of these queues see Figure 4.5.

5. There are three different kinds of acknowledgements in the system. A `rack_data` is an unreliable acknowledgement of the receipt of a data packet, a `dack_data` is a reliable acknowledgement of the delivery of a data packet, and a `rack_dack` is an unreliable acknowledgement of the receipt of a `dack_data` packet.

Figure 3.6: Additional explanations of the verbal event graph in Figure 3.2 and its edge conditions (Figure 3.5).

These parameters can be divided into three logical categories: sender and receiver characteristics, sending behavior, and protocol characteristics. Sender characteristics include an overall send window, a process-process window (the two-tiered window system we implemented uses two windows), and timeouts for a retransmit timer and propagate stability timer (stability information is delayed in the hopes of piggybacking it on other messages; this timer is activated if it is not possible to do so). Receiver characteristics include a receive window and timers for acknowledgements and requests for stability updates (both types of control messages may be delayed in the hopes of piggybacking them on other messages). Sending behavior is controlled by a method of sending [randomly according to a Poisson distribution (with some rate λ), on sending a message elsewhere, on reception or delivery of a message, or at a fixed interval], packet size, burst size (length of message sent; it can be randomly distributed), and type of multicast used (hardware or software multicast). Lastly, protocol characteristics identify the ordering protocol used and specifies, in the case of the conservative protocol, the desired degree of stability desired (the k in k-stable).

In general, we will assume that processes are homogeneous. This means that the general characteristics of all processes in a system are the same. They will differ only in their sending behavior: some will primarily be packet sources, some will primarily be packet sinks, and some may act as packet forwarders.

A `host` component simulates a machine interface between processes and the network. This interface does not include transport level protocols such as flow control and error detection. Such protocols, as described above, form a part of the process component. The host component primarily acts as a messages buffer. In particular, this means that a host's behavior is defined by a delay, and input and output queue sizes. The delay is meant to simulate the delay caused by the transfer of a message from user space (in the process) to the wire and was experimentally determined to be 125 μs (Sparc 1^+) [van92]. These parameters will vary depending on the machines used and on the interface to the network. In the simulations done here, infinite sized queues were used for the host machines. This was done because the goal of this work was to identify how frequently messages arrive out of causal order. Clearly, when internal buffers overflow and packets are lost, messages will be more likely to arrive out of order. Infinite buffers allow us to identify those situations in which messages arrived out of order for reasons other than retransmissions.

`group` components merely retain information about which processes are members of the group. Their only functionality is to replicate packets for each recipient. When hardware multicast is used, this is done immediately prior to reception and when software multicast is used, this is done prior to being placed on the network.

The local area network is modeled by a 1-persistent, non-slotted network on which the stations are uniformly positioned. Instead of using a random back-off scheme for retransmitting packets which collide, the network queues colliding packets and retransmits them as soon as the network is free. Such a capability does not really exist in an Ethernet. The consequence of this assumption will be to make the model more efficient than a real Ethernet. Throughput will be higher and delays smaller. However, this will serve to further minimize the number of packets delayed at reception and to illustrate the magnitude of the measured delays caused by the ordering protocols. The primary parameter used to define the network is the speed of the network. In the experiments described in this chapter, this is 10 Mbits/sec.

3.2 Experiments

The family of protocols that were studied range from the optimistic protocol to the conservative protocol. These protocols maintain causal order by delaying packets — at send and/or at delivery — and by attaching ordering information, in the form of vector timestamps, to the packets. Their performance is expected to differ because the optimistic protocol attaches a vector timestamp for each group in the system to each message, while the conservative protocol attaches only a timestamp for the destination group. This lowers the overhead of the conservative protocol, but means that it must use additional delays to guarantee order: the sender may be delayed until it is known that all messages to, or from, other groups are known to have arrived at all members.

This section looks at the questions posed in the chapter introduction: how frequently do multicasts arrive out of causal order and what is the "cost" of out of order packets? They questions are discussed in further detail and experiments designed to measure them are presented. Finally, Section 3.3 provides the results of these experiments.

3.2.1 Frequency of Out of Order Packets

Packets can be expected to arrive out of causal order when the potential for causal dependencies exists. It is expected that in the single local area network situation investigated here, the largest number of out of order packets will occur in very simple situations. A scenario which is expected to be a worst case situation is depicted in Figure 3.7. This scenario involves 3 overlapping process groups. Messages will arrive out of order in the following situation: Suppose process p_1 sends message m to group G_3 and then message m_1 to group G_1. On reception of m_1, process

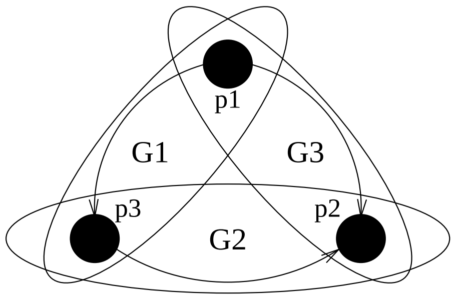


Figure 3.7: Worst case situation for counting frequency of out of packets arriving out of order.

p_3 sends message m_2 to group G_2. Message m_2 will arrive out of causal order if it arrives at p_2 prior to the initiating message m. The number of misordered packets can be increased by increasing the number of processes in the intersection of groups G_2 and G_3 (by replacing process p_2 with a set of processes).

3.2.2 Actual Cost of Maintaining Order or Delay Cost

In the optimistic protocol a delay occurs only after reception and prior to delivery at a destination process. The vector timestamp information is used to deduce whether or not a message is out of order. If it is, a delay occurs until any causally preceding messages have been received and delivered. In the conservative protocol, the single vector timestamp contains insufficient information to deduce whether there are causally preceding messages from other groups that are missing. The single vector timestamp contains only enough information to order messages within a single destination group. In order to deal with messages from other groups arriving out of order, the conservative protocol delays sending messages until it is certain that no such messages can exist. That is, it waits until any messages it has sent or received from other groups are stable (they have been received by all destinations). The conservative protocol is therefore expected to delay less frequently at delivery because of out of order messages within the same group and to delay at send when messages have been sent or received from other groups. The amount of delay will depend on how frequently each of these events occurs.

In order to measure the potential size of the delay, scenarios where long delays could be expected were studied. These scenarios are depicted below in Figure 3.8. In case (a), packets flow from the rightmost group to the leftmost groups.

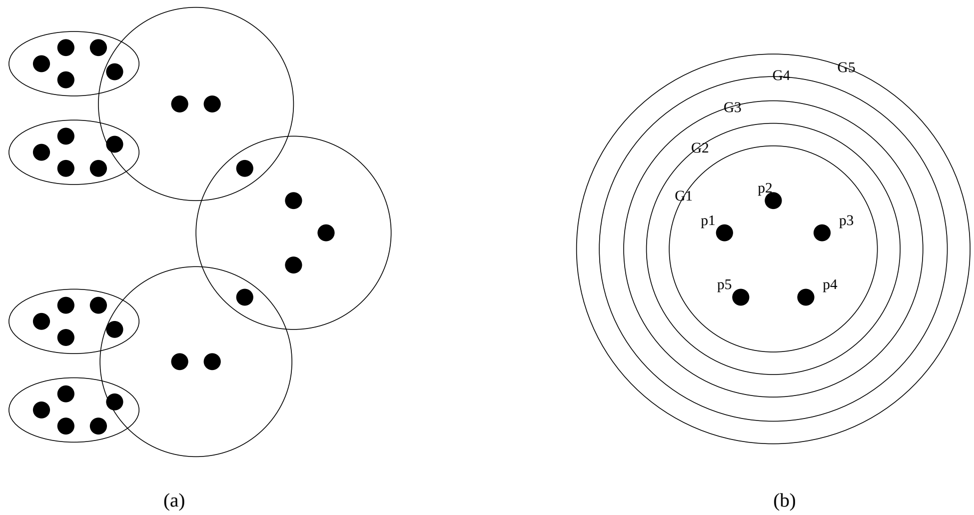


Figure 3.8: Worst case situations for ordering delays.

Processes in the intersection of 2 or more groups send upon reception of a message from its right neighbor. This means that nearly every send results in a delay for the conservative protocol. In case (b) each process sends randomly to all groups. Consequently, there is a delay for most sends (there was likely an immediately preceding send to another group).

3.3 Experimental Results

Each packet, as it is created and travels through a defined system configuration, is annotated with information representing the delays it incurs. That is, it is annotated with a creation time, the length of the delay due to the ordering protocol prior to sending, the time before it is received at each destination, and the delay due to the ordering protocols before delivery at the destination. Delays at delivery may caused by a halted process (there are delayed sends) or by a packet is received out of order.

This data can be collected in one of three ways: the data for each packet can be recorded, the data can be summarized as a collection of probability distributions, or the data for a collection of packets can be batched together and summarized. This summary can then be recorded. The first approach to data acquisition, while it provides the most detailed description of an experiment, results in very large quantities of data being collected — typically hundreds of megabytes for each experiment. This method is therefore used only occasionally to examine all delays in detail. The method of probability tables is used as a basis for the work of later chapters. Summarizing the data suffices for most of the purposes of this chapter. Using summaries of batches of data provides a means to efficiently determine confidence intervals for the expected values of both delays and throughput.

The data summary collected for each batch includes the total number of packets sent, the number of packets delayed at send, the cumulative length of this delay (over all packets in the batch), the cumulative time from send to receipt (over all packets in the batch), the number of packets delivered, the number of packets delayed because of a halted process, the cumulative length of this delay (over all packets in the batch), the number of packets delayed because of out of order reception, and the cumulative length of this delay (over all packets in the batch).

A combination of batched data and of replication is used to compute 95% confidence intervals for experimentally determined expected values of the throughput, the proportion of packets delayed by the ordering protocols, and the lengths of these delays. In general, the size of each batch depends on the amount of data (which in turn depends on the applied load and resultant system throughput) and the number of data recorders in a given experiment. The number of batches depends on the length of the experiment; batches typically summarize some fixed interval of simulated time.

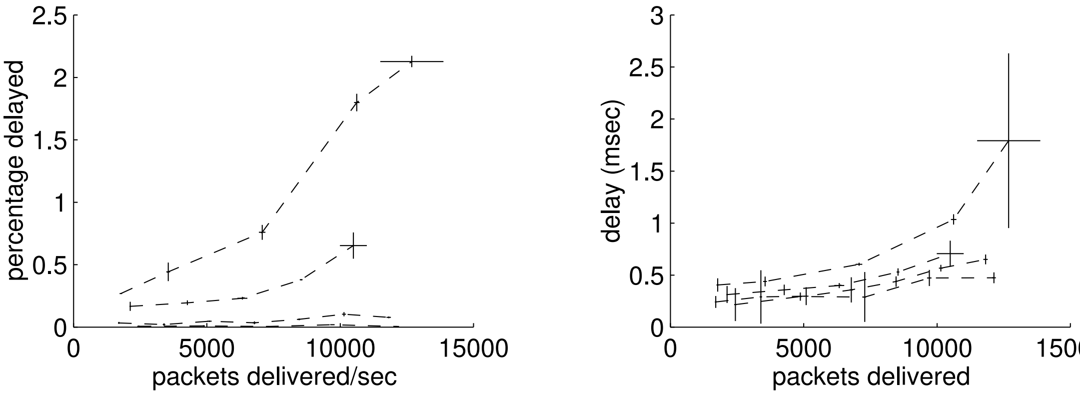
Initialization bias was not found to be an issue in the simulation experiments. There is a small period of initialization when queues are filled with packets. Any bias caused by this initialization is overwhelmed by additional data. Initialization time is typically much less than 1 second of simulated time. Experiments were typically run for between 100 and 300 seconds of simulated time.

3.3.1 Frequency of Out of Order Packets

Estimates of how many packets arrive out of order for the scenario in Figure 3.7 are presented in Figures 3.9 and 3.10. Recall that packets from process p_3 may arrive out of order at processes in the intersection of groups $G2$ and $G3$. The proportion of out of order packets is expected to be greater when the number of processes in this intersection is larger. Experiments were therefore run with between 1 and 21 processes in the intersection of the groups $G2$ and $G3$. Figures 3.9 and 3.10 show the percentage of delivered packets that were delayed at reception (because they were out of order) and the average length of the delay incurred for each delayed packet. Shown are plots for 9, 13, 17, and 21 members in the intersection of groups $G2$ and $G3$. These figures show that, in general, the greater the load applied to the system (reflected in the number of packets delivered; the x-axis), the more likely it is that packets arrive out of order. The two figures differ in the order in which they send packets to group members. In Figure 3.9 both senders (p_1 and p_3) send to members of $G2 \cap G3$ in the same order. In Figure 3.10 each sender sends to members of $G2 \cap G3$ in a different order; in this case, the reverse order.

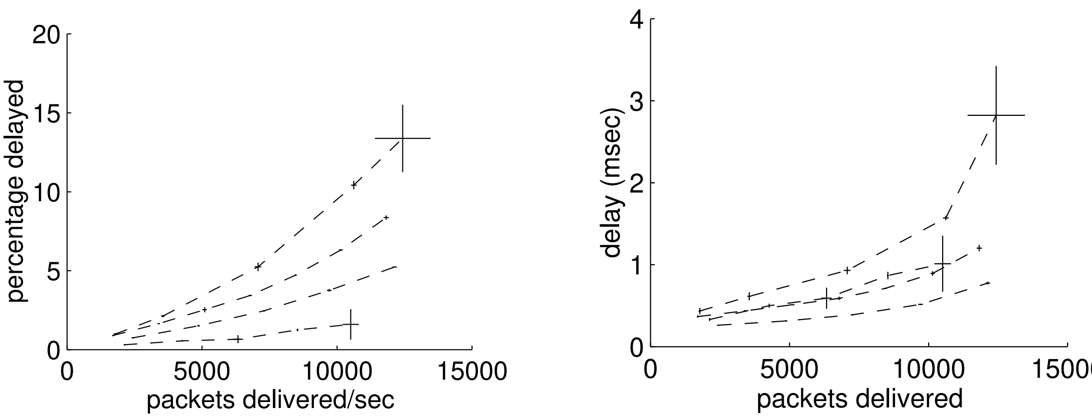
The proportion of packets that arrive out of order is much greater when the group members are ordered in different ways. This occurs because the delay for p_2 to send to the last process in the list is likely to be much higher than to send to p_3 and have p_3 send to its first destination (which might be the last process to which p_2 is sending). We note, however, that despite an approximate five-fold increase in the number of packets delayed, the cost of an average delay increases only approximately two-fold. This occurs because the delays remain sufficiently randomly distributed that they do not interfere with each other. This issue is further explored in Figures 3.11 and 3.12. In these figures, the length of each delay is plotted as a function of the absolute time at which the delay occurred. Further, a histogram is presented that shows the percentage of delays that were of a given duration. In each case, the intervals in the histogram are 0.1 ms. In the scatter plots, vertical stripes of points, for example, at 50.64 s and at 72.95 s in the lower plot of Figure 3.12, indicate a sequence of delays that occur when a sequence of packets is delayed at the same process (or at a small set of processes).

Finally, we note that the percentage of packets that arrive out of order and



(a) (b)

Figure 3.9: (a) The percentage of delivered packets that were received out of order and (b) the average duration of the delay thereby incurred. The five lines are for 9, 13, 17, and 21 members in $G2 \cap G3$. The two sending processes send to the intersecting processes in the *same* order.



(a) (b)

Figure 3.10: (a) The percentage of delivered packets that were received out of order and (b) the average duration of the delay thereby incurred. The five lines are for 9, 13, 17, and 21 members in $G2 \cap G3$. The two sending processes send to the intersecting processes in *different* orders.

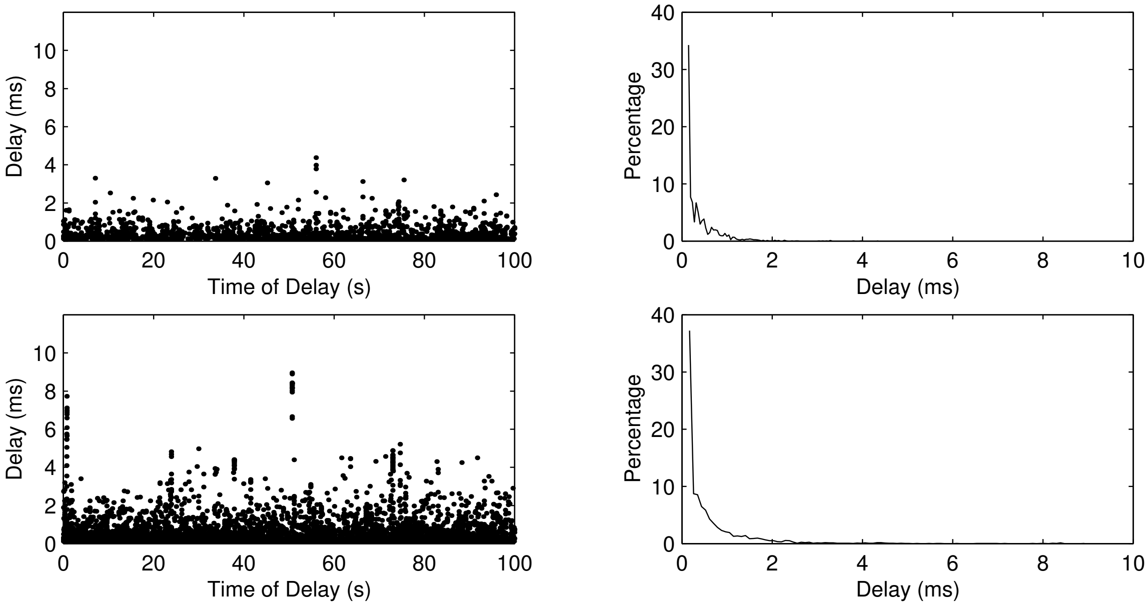


Figure 3.11: Detailed description of the delays incurred by packets arriving out of order for the case with 17 members in $G2 \cap G3$ and with two different loads on the system. The two sending processes send to the intersecting processes in the *same* order. The scatter plots on the left show the length of each delay as a function of the absolute time at which the delay occurred. The histograms on the right show the percentage of packets experiencing delays. The histogram is divided into 0.1 ms intervals.

the length of the delay incurred by out of order packets increases superlinearly as a function of the load on the system. This may be caused by saturation of the network which would result in larger delays and hence in an increased number of misorderings. If we consider a vertical cross-section of Figures 3.9 and 3.10, it is not clear whether or not the percentage of packets that arrive out of order and the length of the delay incurred by such packets increases superlinearly as a function of the group size. It does for the results presented in Figure 3.9, but not for those presented in Figure 3.10.

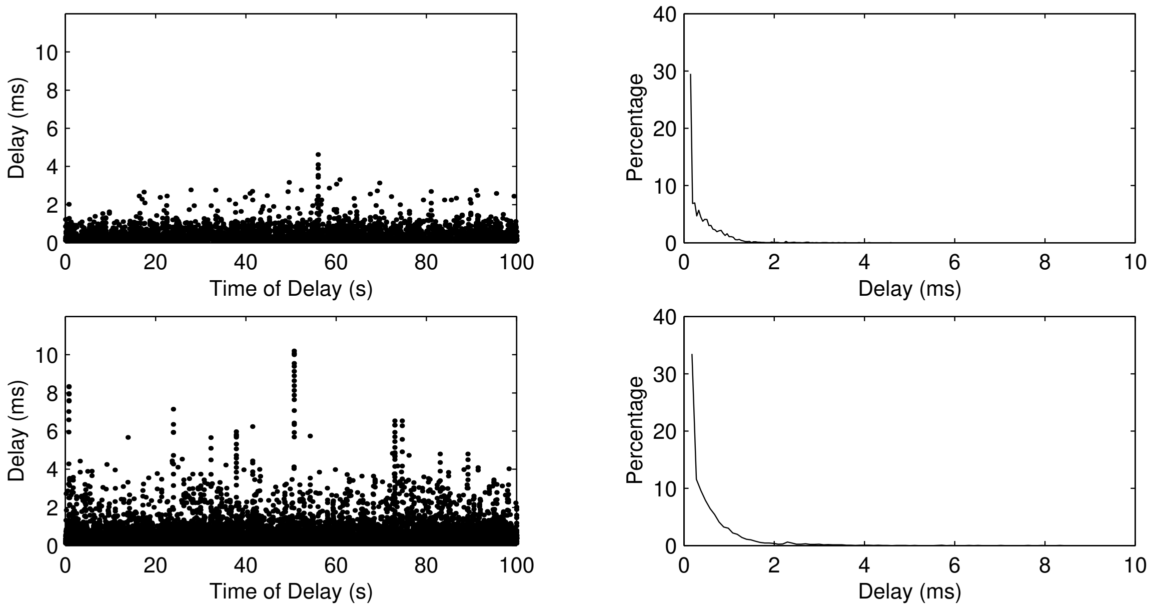


Figure 3.12: Detailed description of the delays incurred by packets arriving out of order for the case with 17 members in $G2 \cap G3$ and with two different loads on the system. The two sending processes send to the intersecting processes in *different* orders. The scatter plots on the left show the length of each delay as a function of the absolute time at which the delay occurred. The histograms on the right show the percentage of packets experiencing delays. The histogram is divided into 0.1 ms intervals.

3.3.2 Actual Cost of Maintaining Order or Delay Cost

Simulation results measuring the delays incurred by the multicast protocols can be found in Figures 3.13 through 3.18. Figures 3.13 and 3.14 illustrate the cost incurred in the hierarchical experiment [see Figure 3.8(a)] and Figures 3.15 through 3.18 illustrate the cost incurred in the overlapping group experiment [see Figure 3.8(b)]. These results are now discussed in detail.

Case (a) — Hierarchical Groups

Figures 3.13 and 3.14 show the average delays associated with the ordering protocols for binary trees built of 5-member groups of heights 2 and 4. Each graph

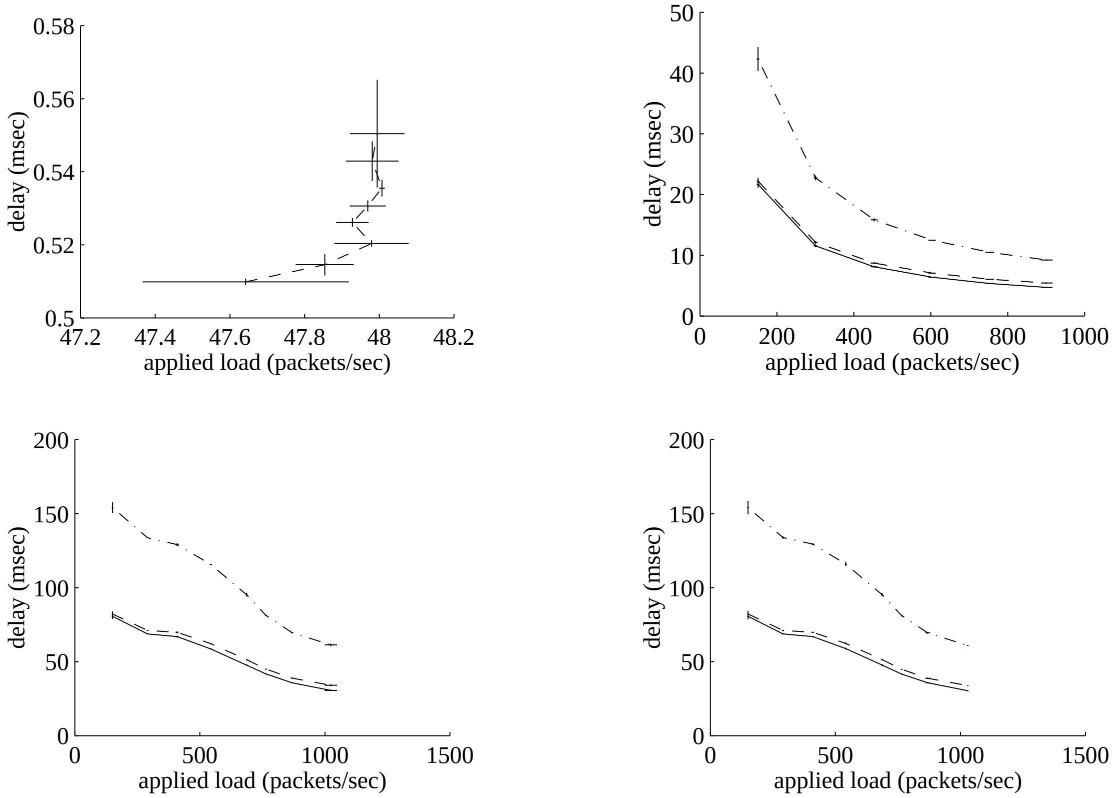


Figure 3.13: Average delays incurred by the 0- (optimistic), 2-, 3-, and 4- (conservative) stable protocols for a hierarchy of groups of depth 2. Delays incurred by the 1-stable protocol are similar to those for the 0-stable protocol. Each graph shows the average delay prior to sending (—), the average cost to send (−−), and the average delay at reception (because the recipient is halted trying to send elsewhere) ($-\cdot-$).

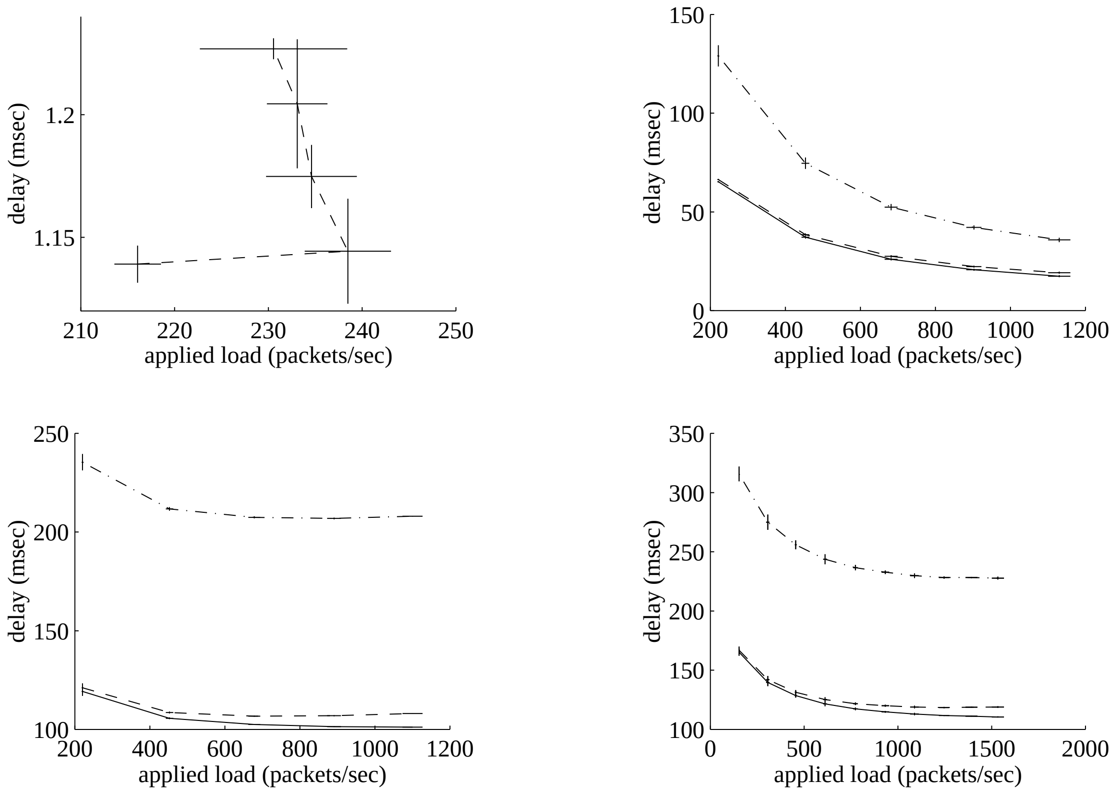


Figure 3.14: Average delays incurred by the 0- (optimistic), 2-, 3-, and 4- (conservative) stable protocols for a hierarchy of groups of depth 4. Delays incurred by the 1-stable protocol are similar to those for the 0-stable protocol. Each graph shows the average delay prior to sending (—), the average cost to send (−−), and the average delay at reception (because the recipient is halted trying to send elsewhere) ($- \cdot -$).

plots three delays: the average delay of a packet prior to send, the (additional) average cost of the send, and the (additional) average delay at reception prior to delivery (because the receiving process was halted trying to send something else). Note that these costs are incurred only by packets sent by processes in the intersection of groups.

The four graphs in each figure correspond to differing stability requirements. The first corresponds to a requirement of no stability; that is, the sender is never delayed until packets become stable. Consequently, it is more likely that packets will be delayed because of out of order reception. The second, third, and fourth graphs correspond to stability requirements of 2, 3, and 4. That is, the sender must ensure that, respectively, 2, 3, or 4 messages of a multicast to (or from) another group are stable before sending a new multicast. In each case, packets may still be delayed at reception because of out of order reception. However, this occurs less frequently when the stability requirement is greater.

In each experiment, a single process in the root group sent randomly distributed messages to the group. Each of these messages was echoed to the children groups by the processes in the intersection of groups. Packets were forwarded as soon as they were received. Thus, every packet, except those sent by the packet originator, was delayed. The pattern of communication explains the poor throughput of the 0 stable case (which we might expect to be the best). Because .ackets are unidirectional, a sender must wait, when its send window is full, for a recipient's acknowledgement timer to expire. In this study, this is a fixed length of time. This communication pattern also explains the decrease in delay as the load is increased. As the load increases, the receive windows are more likely to be full. Therefore acknowledgements are more likely to be more timely—when a receive window is 75% full, acknowledgements are sent. Shorter delays result. Adaptive timeout algorithms would reduce these effects. Finally, we observe that as stability requirements increase, the average delays generally increase and that the average delay at packet reception is approximately the same as the average sending delay.

Because the experiments were designed to delay (nearly) every packet, the delays should be interpreted as the cost incurred by those packets which are actually delayed. In particular, this means that messages composed of more than one packet will typically incur this cost only once for the whole message and that many very small messages might incur a lower cost if they can be packed into a single message.

Case (b) — Overlapping Groups

Figures 3.15 through 3.18 show the average delays associated with the ordering protocols for overlapping groups of 3 and 7 members. Each case contains a different number of overlapping groups. These are: 3 overlapping groups of 3 members (Figure 3.15), 5 overlapping groups of 3 members (Figure 3.16), 3 overlapping groups of 7 members (Figure 3.17), and 5 overlapping groups of 7 members (Figure 3.18). As in the hierarchical case above, each graph plots the average delay of a packet prior to being sent, the average cost of reception, and the average delay at reception. Further, each of the figures contains graphs for stability conditions ranging from 0 to the size of the group (less 1; a message sent to oneself is already stable). The results shown here are representative of similar experiments done for 2 through 9 groups of from 2 to 7 members.

In each case, when no stability requirements are made, the cost is low and throughput is high. The low cost results because no sends are delayed. The high throughput occurs because the timeout phenomenon observed in case (a) does not occur here. It does not occur because data packets are sent in both directions, whereas in case (a) they were sent in only one direction (from the root of the tree to its leaves). Acknowledgements are piggybacked on all data packets resulting in faster acknowledgement. This prevents the full send windows that occurred in case (a).

When 1-stability is required, delays drop with larger loads. As in case (a), delays drop because higher loads mean that it is more likely that information is passed back to a sender sooner — there is less chance that the sender needs to request it. Further, a feedback mechanism occurs by which the lower costs result in fewer packets being delayed. This, in turn, reduces the delays. As the load increases, this effect decreases; ie, more packets are delayed.

For all other levels of stability, the cost is very high. Nearly every packet is delayed — greater than 97% of all packets. Consequently, each sender spends large periods of time waiting for stability information. Further, sends become very "bursty". That is, when a sender is granted permission to send, many delayed sends occur at once. A result of this is that the potential benefit of piggybacking

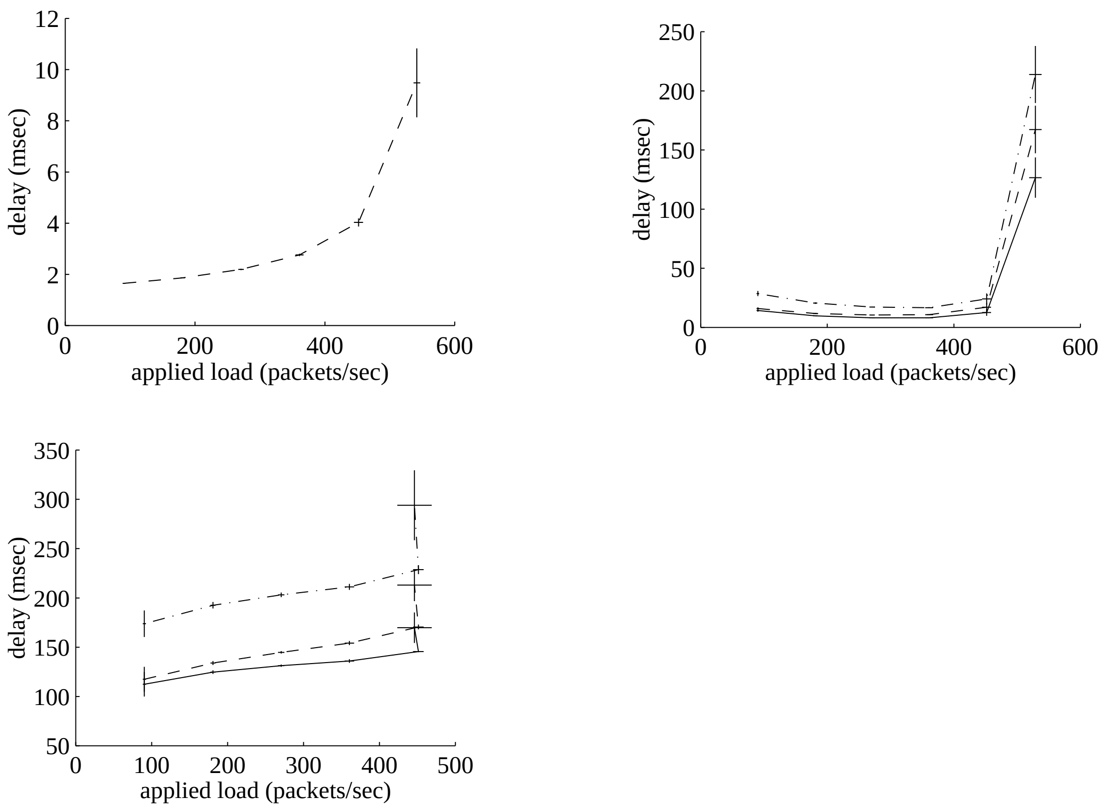


Figure 3.15: Three groups containing the same three processes. Each graph shows the average delay prior to sending (—), the average cost to send (−−), and the average delay at reception (because the recipient is halted trying to send elsewhere) (− · −).

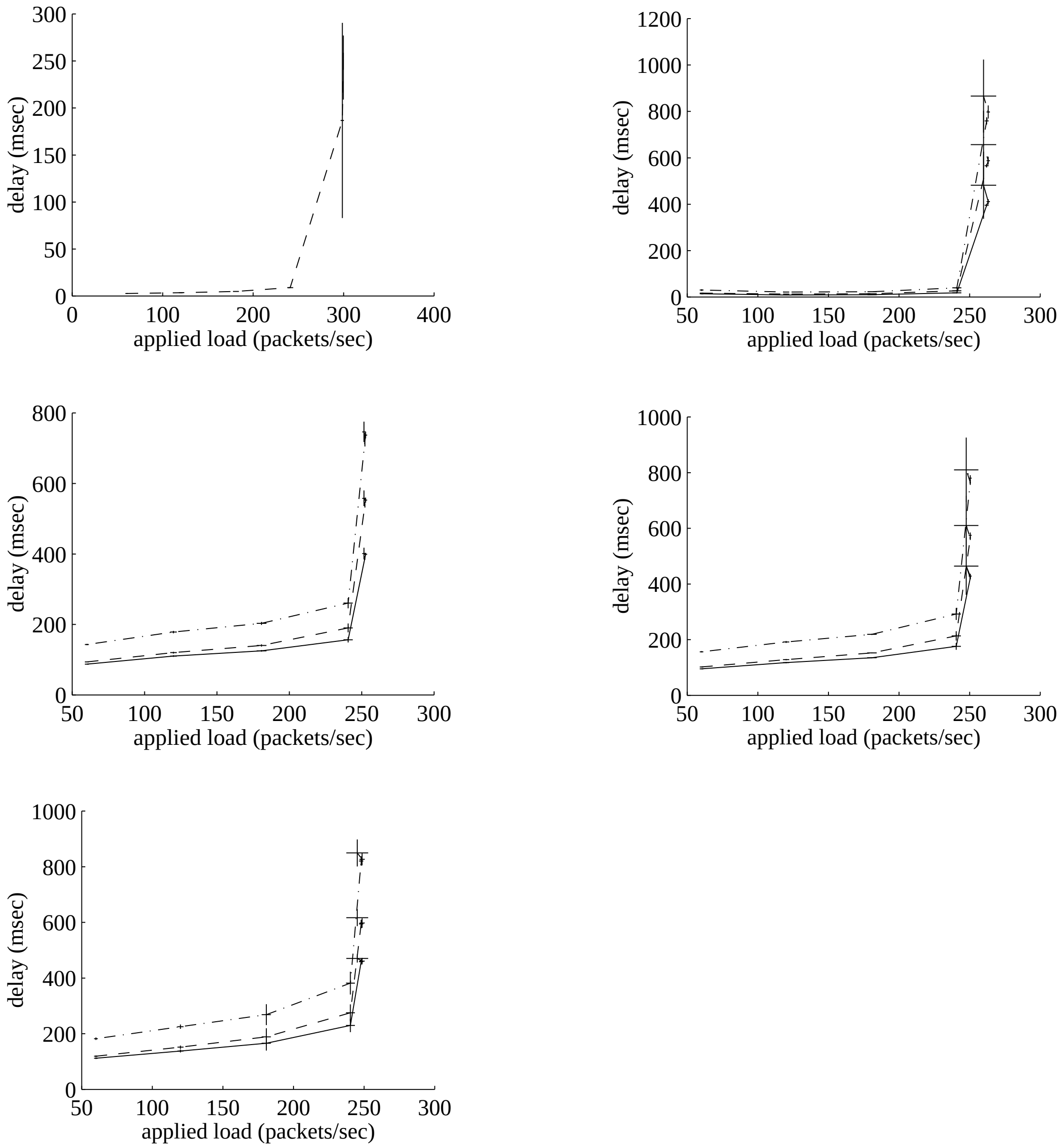


Figure 3.16: Three groups containing the same five processes. Each graph shows the average delay prior to sending (—), the average cost to send (– –), and the average delay at reception (because the recipient is halted trying to send elsewhere) (– · –).

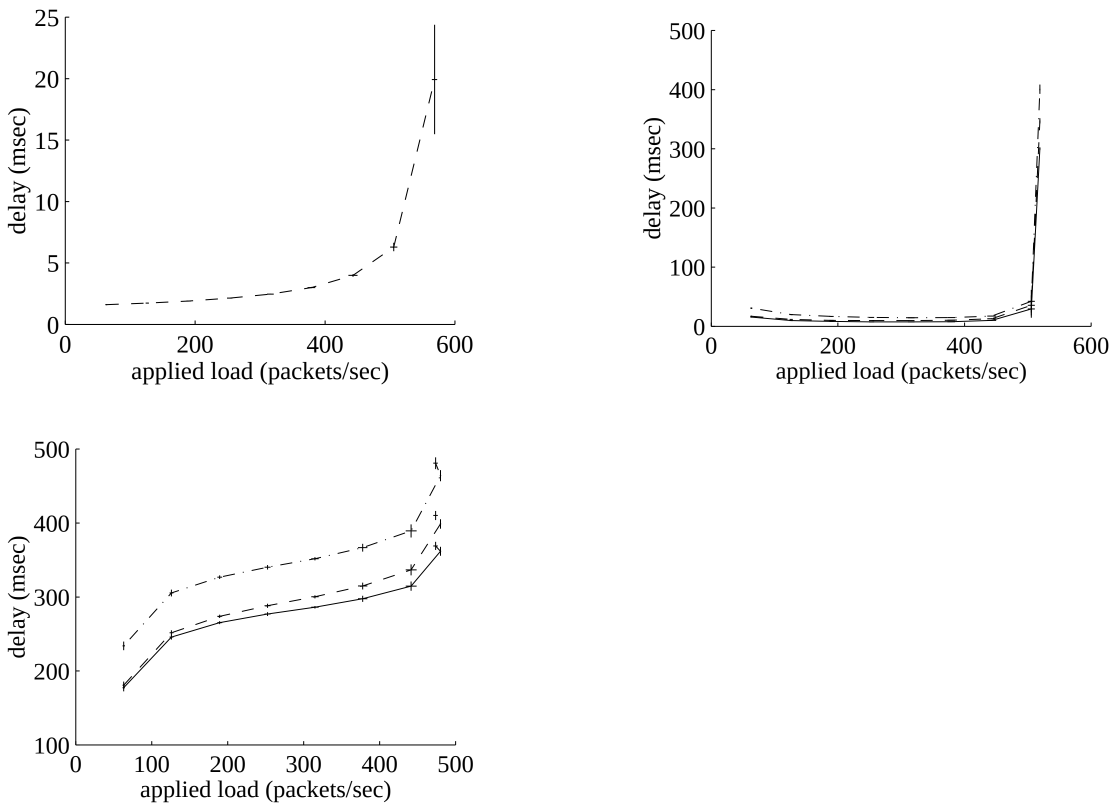


Figure 3.17: Seven groups containing the same three processes. Each graph shows the average delay prior to sending (—), the average cost to send (−−), and the average delay at reception (because the recipient is halted trying to send elsewhere) ($- \cdot -$).

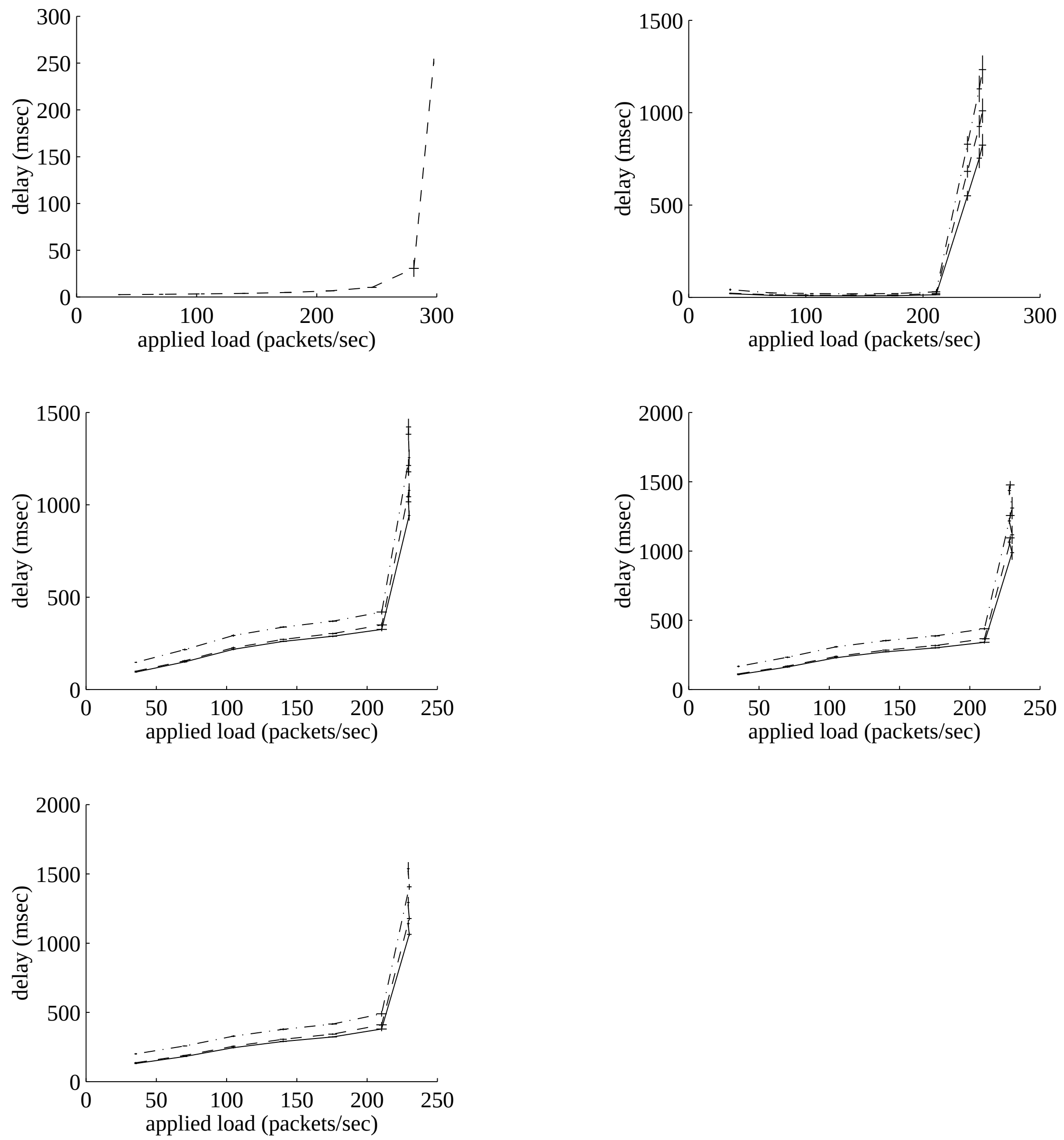


Figure 3.18: Seven groups containing the same five processes. Each graph shows the average delay prior to sending (—), the average cost to send (−−), and the average delay at reception (because the recipient is halted trying to send elsewhere) (− · −).

information on data packets is minimized: information is rarely piggybacked. This burstiness is further explored in Section 4.6.

3.4 Discussion

3.4.1 Frequency of Out of Order Packets

The simulation provides a pessimistic estimate of the number of packets that are received out of order. That is, the results underestimate the number of out of order packets. This underestimation occurs for several reasons. First, the network model does not simulate random backoff when a collision is detected. The expected transmission time for packets is therefore lower. Hence, there is a lower probability that causally succeeding packets will arrive first. Second, the simulated environment is limited to a single LAN. Packets forwarded to geographically distant LANs could experience widely different transmission times because of higher fault probabilities, overloaded intermediate gateways, longer and/or slower links, and so on. Packets taking different routes between LANs may well arrive out of order if their relative transmission times are widely variant. In the single LAN case, all the delays are small, reducing the number of out of order packets. Third, the model does not take into account all the potential sources of lost packets. It considers only packets lost at reception because the destination machine could not process packets sufficiently fast. In fact, the loads considered here do not cause this effect. Further causes of packet loss would result in many more packets arriving out of order — the retransmission timeout is very long compared to all other delays. Fourth, messages used in the experiments were a single, small packet in length — 64 bytes. Longer messages (megabytes in size) would take more time to reach their destination (they may be very much delayed if sending windows are too short for the whole message) and would cause shorter, causally subsequent messages to arrive earlier (out of causal order). Finally, each host machine in the simulation is dedicated to serving the sending/receiving process. That is, no delays are caused by a sending or receiving process that is swapped out. The magnitude of this effect is not known. It has been observed in experiments with high speed networks [Jai94]. Despite these pessimistic assumptions, large numbers of packets can be delayed — up to 15% in our experiments. We showed, however, that this number is highly dependent on the order in which a process sends packets to a group. Our results suggest that group membership information should be ordered.

3.4.2 Actual Cost of Maintaining Order or Delay Cost

The simulation shows that, in a fully conservative protocol, delays, when they occur, are very expensive — orders of magnitude greater than in the case where no delays occur. Further, a delay of similar magnitude exists for k-stable protocols where $k \geq 2$. Surprisingly, the cost of the delay drops significantly for $1-$stable protocols. This occurs because of a feedback mechanism that reduces the number of delayed packets. This, in turn, reduced the delays.

These experiments have attempted to maximize the number of delayed packets. The principle of locality [Mog92] suggests that in real applications, a smaller percentage of packets will be delayed. Further, techniques such as packing many small messages into a single larger one would lead to smaller amortized costs. Nonetheless, the simulation results show that when the delays occur, they are very expensive.

Finally, our experiments suggest that if it can be assumed that no more than 1 failure can occur (during the time for 1 message to become stable), 1-stability should be used. However, a point ignored in the simulations is the needed content of the vector timestamp. Because full stability is not being required, a single vector timestamp does not suffice to maintain order. In fact, the full set of vector timestamps (as in the case for no stability) is needed. The gain is in the ability to handle a single failure.

3.4.3 Scalability of the Simulation Model

The small experiments described here have shown the simulation model to be too inefficient for the more ambitious goal of studying the behavior of a large distributed systems. The cause of this insufficiency is an inability of the model to scale to large numbers of processes and large numbers of groups. In order to investigate the behavior of larger systems, it will be necessary to refine the simulation model. It is proposed that a reasonable method would be replace some computations by random functions experimentally determined using the current model. Further, the current simulation considers all processes in the system. It suffices to consider only those processes that are senders and those recipients that serve as "data collectors." Other recipients serve only as data sinks and can therefore be eliminated from the model. These modifications to the simulation model are investigated in the next chapter.

3.5 Conclusion

In this chapter, a model of process group systems was developed. It was used to estimate the number of packets that arrive out of order and their delays. For small groups, relatively few messages arrive out of order. Further, the delays incurred by such messages are small: typically a few packet transmission times. This is true until a very large percentage of packets arrive out of order. Both the percentage of messages that arrive out of order and the delay they incur seem to increase superlinearly with the load applied to the system. Finally, significantly more packets will arrive out of order when senders maintain membership views in which members are sorted differently. A membership service should therefore maintain ordered membership lists.

Additionally, the model was used to estimate the costs of the conservative protocol. It was found that when delays are necessary they are very expensive — orders of magnitude greater than the time to transmit a message using the optimistic protocol. It was also found that these costs are just as high for the k-stable approximations of the conservative protocol where k is 2 or more. When k is 1, the cost may be high, but not as high.

While the simulation model developed worked well for the small experiments done in here, it proved too inefficient to be used for large systems because it maintains (and must update) too much state. A refinement of this model is developed in the next chapter which can then be applied to a large system.

Chapter 4

Group-Based Simulation Model

Initial investigations using a process-based simulation model were described in Chapter 3. Unfortunately, the process-based approach is unsuitable for studying large models because it is too computationally intensive. This results from the detail of the model. The high cost makes the model unusable for simulating the large systems we wish to study. A refinement of the simulation model is therefore developed in this chapter. This refinement is *group-based*: groups are the basic entities rather than processes. The process-based model is used to investigate those aspects of a single group which best characterize group behavior.

This chapter motivates the group-based model, describes it, identifies its key parameters, and presents some initial experiences with it.

4.1 Motivation for Model Refinements

Experience with large group-based distributed systems has suggested that several basic group structuring paradigms are commonly used [BCG91]. Two of these, hierarchical groups and peer groups, provide the motivation for the model refinements described in this work.

The first of these paradigms is common in applications in which large amounts of information must be reliably transmitted from a small number of sources to a large number of destinations. For example, information generated on the trading floor of the stock market must be transmitted both to viewing screens on the floor and to many recipients in brokerage houses. This can be done in an hierarchical manner such as in Figure 4.1. At the bottom level of the hierarchy large diffusion

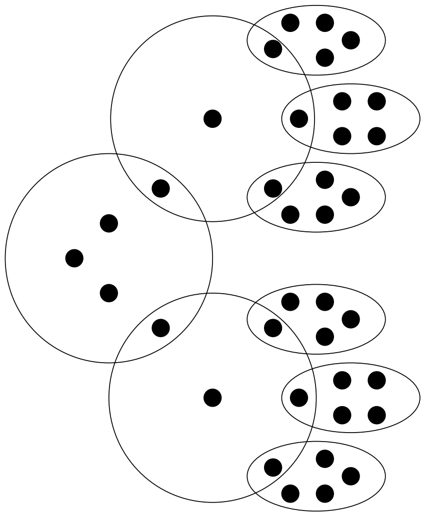


Figure 4.1: Hierarchical structuring of process groups used for information dissemination.

groups [BCG91] may also be present. The ISIS "News" tool [Gla95] hierarchically structures groups in this way to disseminate large amounts of information.

A second class of applications use groups for process[1] control, such as a factory floor. In such settings many small, and relatively independent, industrial processes exist. Each process would be controlled by a group to ensure reliability and fault tolerance of the process (including the failure of the machines involved in the process). Groups controlling related processes may overlap. Finally, there would be an overall control group linking all the other groups together. Such an application is depicted in Figure 4.2.

These two common classes of applications reveal that many similar groups are used in large systems. Further, these groups are tiled in a regular way. The behavior of processes in "similar" groups is expected to be similar. It should therefore be unnecessary to model the behavior of every process in the system.

[1] Note that the use of the term process here refers to a "method", or "a series of actions or operations conducing to an end," not to a process executing on a computer.

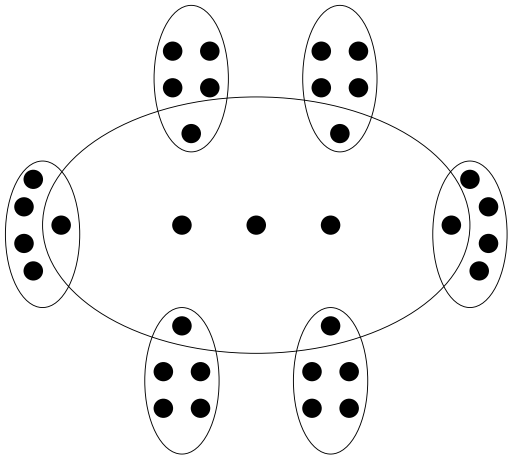


Figure 4.2: Typical group structure for process control applications.

It suffices to model groups as entities or to model only some of the processes in the system. The next sections discuss the method used to simplify the simulation model so that it would be both efficient and effective.

4.2 Overview of Group-Based Simulation Model

Recall, from Chapter 3 (Section 3.1), that we can view a system of process groups as a directed multigraph $CS = (P, E)$. There is an edge from p_1 to p_2 in the graph whenever p_1 sends multicasts to p_2 (via some group g of which both are members). The process-based model implements this view of a system by simulating every process, the host machines on which they are resident, and the network. Further, every packet in the system is traced from creation, until it becomes stable (and so can be destroyed).

As suggested by the discussion above (Section 4.1), a first approach to simplifying the simulation model would be to model groups as entities. That is, we can redefine the graph to be $CS' = (G, E')$ where the nodes G correspond to process groups and an edge $< g_1, g_2 >$ is in E' if and only if there exists a process p in both groups g_1 and g_2 (CS is an undirected graph). CS' is often called the *communication structure* of a system [BSS91]. An example of this representation is found in Figure 4.3.

This first attempt at simplifying the model is too great a simplification. It

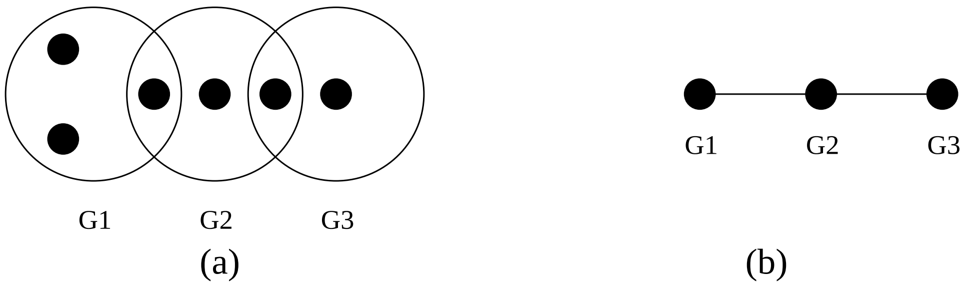


Figure 4.3: Example group structure and its corresponding communication structure.

necessitates the assumption that each member of a group behave in exactly the same way as every other member of the group. This is an impractical assumption to make when some members of a group are members of more than one group. Such group members are is some sense "special" in that they serve as collectors, transmitters, and emitters of external information. Consequently, it will be desirable to allow them to send messages at a rate different from other members of the group. It will also be desirable to allow such members to send at rates different from each other; that is, the group members who are in more than one group need not send at the same rate as each other.

In addition to appearing to be too inflexible a simplification, this method of modeling a group would prevent a study of the delays of ordered multicast because it entirely eliminates processes from the model. It therefore prevents any study of the delays that occur at individual processes. All that can be studied are estimations of the cost based on the group itself. How these costs would vary depending on, for example, group overlap would be difficult to determine in a meaningful way.

A second approach is to consider modeling some, but not all, processes in the system. The question is: which ones? One idea would be to model only "unique" processes. That is, processes that behave identically would be modeled by a single process. This may be impractical because it isn't possible to easily identify which processes will behave identically. In general, this will be a function not only of the processes, but also of the system structure and communication pattern. A second idea, and the one adopted here, is to model those processes which act as sources and transmitters of data. That is, those processes which are either sources of data intended to be sent outside the group and those that are in the intersection of groups (the transmitting processes). It will also be convenient to include some data sinks in the model; these are used to collect simulation statistics. This idea can be modeled by redefining the graph CS to be $CS'' = (P'', E'')$ where $P'' \subseteq P$ contains those processes p in the intersection of at least two groups, and perhaps a subset of the others, a set of packet sources and packet sinks, and where $E'' \subseteq E$ is the restriction of E to P''. Figure 4.4 illustrates this redefinition.

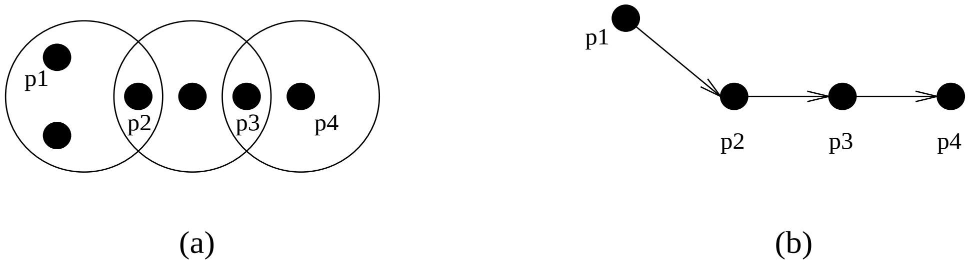


Figure 4.4: Example group structure and its associated graph CS''. In this case, the processes pass information from p_1 to p_4 but not in the other direction.

This second technique for simplifying the process-based model suffers from being more complicated than the first technique. It eliminates those processes that are of less interest because they contribute only to the "background noise" of a group and retains those processes which are directly related to the cause and costs of misordered packets — those processes in the intersection of more than one group.

The following sections discuss in detail how this simplification of the process-based model was implemented.

4.3 Model Assumptions

We assume, for simplicity, that members of a group are (mostly) homogeneous. In particular, we assume that their buffers, including sending windows, are the same length, they send "background noise" to the group at the same rate, and their timeout algorithms are the same. A process which is a member of more than one group differs in that it may send additional packets. Lastly, we assume that all members of a group are on the same local area network. We do not assume that every group is on the same network; that is, processes in the intersection of groups may be connected to more than one network.

We further assume that a group's use of the network is not affected by other groups in the system. That is, the load caused by all processes external to a group is considered to be negligible. The network cost caused by members of the same group will be determined by the process-based model. That the network is otherwise free for use is a simplification that can be explained either by an assumption that every group is on a different network or that the network architecture is not a shared medium. (For example, it might be a packet switched network such as an ATM network.) A consequence of this assumption is that transmission costs will be lower and throughput will be higher than they would be in a shared network. Measured ordering costs will therefore be lower than in a shared network.

4.4 Sources of Process-Based Simulation Cost

The process-based simulation model has a high cost for two reasons. First, there is high overhead associated with each stage of the path that a packet takes. Second, the process-based model includes all processes no matter their "interest", or influence, on the simulation results. These costs are studied and strategies for reducing them are proposed.

Briefly, the path a packet takes after packet creation is (see Figure 4.5): enter a delayed queue if the sending process is halted awaiting a stability condition or if the send window full; when free to send, piggyback any desired information; send to host machine (queuing if busy); upon release from host machine, send to (virtual) group where packet is split and queued if process-to-process window full; send to network (queuing if busy); send to recipient host (queuing if busy); upon release from recipient host to recipient process, enter received queue if not immediately deliverable (out of order); enter delayed queue if process halted awaiting a stability condition; enter delivered queue when delivered but not yet stable; and destroy it when it becomes stable.

Each stage in this path may correspond to a new simulation event or may result

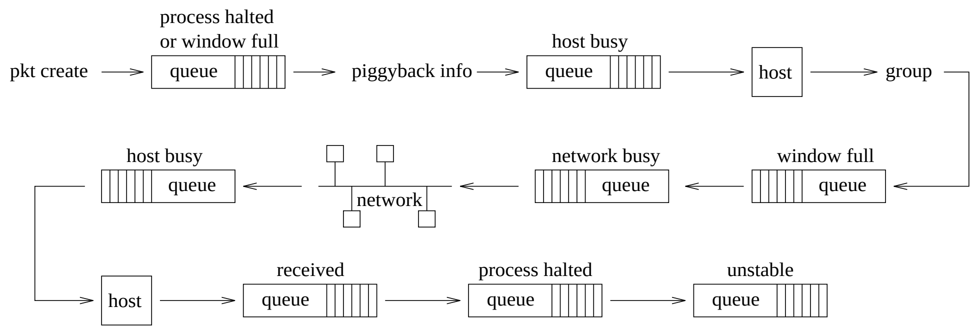


Figure 4.5: Path taken by a packet in process-based simulation model.

in a search of one of the queues. In the first case, the event queue, a record of all scheduled (future) events, is expected to be large because of the large number of packets in the system. It therefore has high overhead — the event queue is maintained in sorted order and is frequently searched for events that need to be cancelled. In the second case, the costs are high because the packet is being used as a key to search whole buffers for related (or identical) entries. In both cases, costs can be reduced by shortening the overall path length of a packet and the lengths of the intermediate queues. One way this can be done is by replacing the cost of sending the packet from one process to another by a random function which estimates the real cost. This random function can take into account the costs of the host machines, the delays caused by full send windows, the network cost, and the transmission cost. Such a random function can be experimentally determined from the process-based model given an assumption about the homogeneity of the processes.

There are two key issues that must be addressed in using such random functions. The first issue concerns identifying which costs cannot be estimated by such random functions — our goal is to measure the costs associated with ordered multicast protocols. Such costs are essentially of two types. These are the cost of delaying a packet at reception (before delivery) because it arrives prior to a causally preceding packet and the cost of delaying a packet at send because it may be received prior to a causally preceding packet. These delays depend on the order in which packets arrive at destination processes and on the time for messages to become stable.

While the delay for a single message to become stable can be computed by a random function, the delay caused by several cannot because it is not known in advance how many such packets there will be. This cost must therefore be computed by the simulation. The second issue is identifying those parameters which best define the random functions. This question is addressed in Section 4.5.

In addition to reducing the length of the path traversed by a packet, the cost of this traversal can be further reduced by eliminating those packets which are "uninteresting"; that is, those that originate from or are to processes not in the intersection of more than one group. Eliminating such processes from the model also removes these packets and all the events associated with them. These processes can be eliminated from the model by appropriately modifying the remaining processes and the random functions suggested above — the additional cost of communicating to those members and of their communication on the group is a function of the size of the group (and should therefore be reflected in the random function).

4.5 Parameterizing Process Behavior

The model refinements described above require defining random functions to compute values that can not be directly computed without all of the processes present. A convenient set of functions compute (a) the time from packet send to packet receive (T_{rcv}), (b) the time from packet send until the the sending process knows it is stable (T_{stbl}), and (c) the time from packet reception until a recipient process knows it is stable (T_{rstbl}). These random functions are clearly dependent on the characteristics of the participating processes (including the host machines on which they reside) ($\mathcal{P}$), the group to which they belong ($\mathcal{G}$), and any fixed parameters like the network and the degree of stability desired ($\mathcal{F}$). They also depend on the number of groups to which, and from which, the sending process sends and receives; that is, the degree of group overlap at the sending process ($\mathcal{D}$) and on the load imposed on the network by other members of the destination group ($\mathcal{L}$). Further, note that T_{rcv}, T_{stbl}, and T_{rstbl} are not independent random functions. For a given packet, $T_{rcv} < T_{stbl}$ and $T_{rcv} < T_{rstbl}$. It will therefore be necessary to compute T_{stbl} and T_{rstbl} as conditional probabilities (dependent on a value for T_{rcv}). In broad terms then, $T_x = f(u, \mathcal{P}, \mathcal{G}, \mathcal{F}, \mathcal{D}, \mathcal{L} \mid T_{rcv})$ where u is a random variable. The parameters $\mathcal{P}$, $\mathcal{G}$, $\mathcal{F}$, $\mathcal{L}$, and $\mathcal{D}$ are now more clearly defined.

We now strengthen the already stated assumption of process homogeneity. In addition to our previous assumptions, we assume that all "uninteresting" processes (those not in the set P''), irrespective of the group of which they are members, are homogeneous. Further, we assume that the host machines and the networks are homogeneous. Thus only processes in the intersection of groups and packet sources may be unique. However, they may differ from other processes only in their sending behavior

The parameters affecting process behavior that we considered are listed in Figure 4.6 This list is simplified by setting some values to constants. For example, packet burst size is set to 1, packet length to 53 bytes (the ATM standard [HHS94]), the host machine delay to 125 μs[2] [van92], and the network speed to 155 Mbytes/sec (the ATM standard[HHS94]). Further, only software multicasting is used. These choices are included in the parameter list (Figure 4.6). In order to determine the importance of the remaining parameters a 2^k factorial design experiment was run using the process-based simulation model. This design is presented below.

4.5.1 Experimental Design

An experimental design is a statistical technique used to determine how various system parameters affect the behavior of a model [LK82]. A design technique identifies the least number of experiments (in this case, simulations) to run to gain the most insight into the parameters of interest. For example, the affect of a parameter could be measured by holding all other parameters fixed and varying the parameter of interest. If parameter i (of k parameters) is varied for n_i different values, this method requires $\prod_{i=1}^{k} n_i$ different experiments. A more efficient design method is the 2^k factorial design. In this method, each parameter is assigned 2 values, a low value and a high value. An experiment is run for each combination of values. The main effect of a parameter is the average change in an output

[2] This is the same number used in the previous chapter. It assumes that no changes, such as those in [BBVv95] have been made to optimize the operating system for a high speed network. Because this figure is high, the effective throughput of the network is actually much lower than would be expected.

1. process parameters (and host) ($\mathcal{P}$)
   - process sending frequency (μ_s)
   - burst size (number of packets) ($b = 1$)
   - packet length ($l = 53$ bytes)
   - overall process send window (w_s)
   - process-to-process send window (w_p)
   - retransmission timeout (t_r)
   - process receive window (w_r)
   - acknowledgement send timeout (t_a)
   - stability request/send timeout (t_{rs}, t_{ss})
   - host machine processing delay ($h = 125$ μs)
2. group parameters ($\mathcal{G}$)
   - group size (g)
3. fixed parameters ($\mathcal{F}$)
   - degree of message stability desired (s)
   - hardware or software multicast ($m = software$)
   - network speed ($n_s = 155$ Mbits/sec)
4. group overlap parameters ($\mathcal{D}$)
   - number of groups to which one sends (O)
   - number of groups from which one receives (I)
5. load parameters ($\mathcal{L}$)
   - frequency of messages sent by others to group (μ_r)

Figure 4.6: Summary of parameters divided into classes $\mathcal{P}$, $\mathcal{G}$, $\mathcal{F}$, $\mathcal{D}$ and $\mathcal{L}$. Those parameters which were assigned fixed values are shown. The list of remaining parameters is reduced by identifying those most influencing the quantities T_{rcv}, T_{stbl}, and T_{rstbl}.

due to changing the value of the parameter from its low value to its high value. The results of a 2^k factorial design can also be used to determine the effects of parameter interactions [3].

Before presenting the experimental design and its results, the outputs of a typical simulation run (experiment) are described. The outputs of interest are probability distributions for T_{rcv}, T_{stbl}, and T_{rstbl}. These distributions are generated by tallying the times for each "interesting" packet present in a given simulation run. A packet is interesting if it originates from a packet source, and is destined for a group, that match the requirements of the input parameters ($\mathcal{P}$, $\mathcal{G}$, $\mathcal{F}$, $\mathcal{D}$, and $\mathcal{L}$). The times are tallied by counting the number of occurrences that fall within predefined discrete intervals. This number, divided by the total number of interesting packets, yields an entry in the probability distribution table. For example, the i^{th} entry in the distribution table for T_{rcv} represents the probability that the time from the send of a packet to its receipt by a destination site is in the interval $[t_i, t_{i+1})$ where t_i and t_{i+1} are known in advance. The distribution tables generated for T_{stbl} (and T_{rstbl}) are two dimensional tables where the $(i, j)^{th}$ entry represents the probability that the time from the send of a packet until it becomes stable (is known by a destination to be stable), given that its time to receipt was in the interval $[t_{r_i}, t_{r_{i+1}})$, is in the interval $[t_j, t_{j+1})$.

The data can be viewed graphically. Figures 4.7, 4.8, and 4.9 are samples of the probability distributions for T_{rcv}, T_{stbl} and T_{rstbl}, respectively. The data for T_{stbl} and T_{rstbl} should be viewed as contour maps. The probabilities are approximated by the different colors and symbols. These are explained by the legend provided. A spike appears at the tail end of the distribution of T_{rcv} (see Figure 4.7). It corresponds to the probability that the time to receive a packet is greater than the maximum predefined interval as described above — because network delays are unbounded, some delays will be very long. Such spikes can be reduced by using a longer (overall) interval. This can be done either by increasing the size of each interval or by increasing the number of intervals (or both). The first approach results in a few very large intervals. Such intervals provide little insight into the nature of the distribution function because each interval is too large: it

[3] Two parameters interact if the effect of one of them depends on the value of the other.

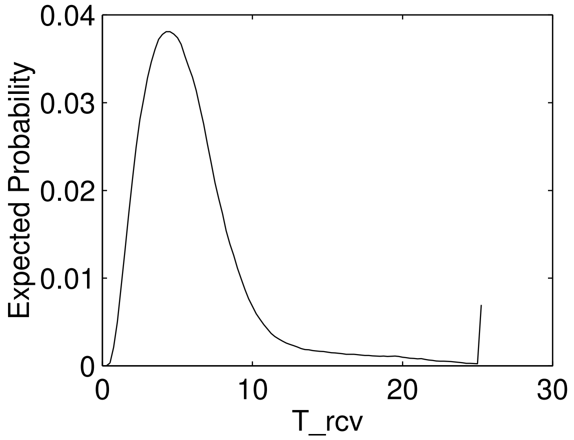


Figure 4.7: Sample probability distribution for T_{rcv} for a run of the process-based simulation. The spike at the end represents the probability that the time to receive a packet was larger than the largest entry in the table.

contains what should be classified as many different values. The second approach results in large distribution tables. The number and size of the intervals used in the experimental design represent a tradeoff between their size, the number of intervals (the size of the resulting probability table), and the length of the overall interval covered.

The amount of data collected will be greater for those runs containing more frequent senders — there will be more interesting packets than when sending frequency is lower. To ensure that the results of each experiment are based on sufficient data, each experiment is composed of 3 independent runs of the simulation model. Each run is for 300 seconds.

4.5.2 Results of Experimental Design

Four steps were taken in the development of the experimental design to identify the most important parameters (among those listed in Figure 4.6) influencing multicast performance.

First, as discussed in section 4.5, a number of parameters were fixed. In general, those that were fixed are dependent on the assumed hardware. Such parameters are ignored in the simulations done here. Studies that look at the influence of

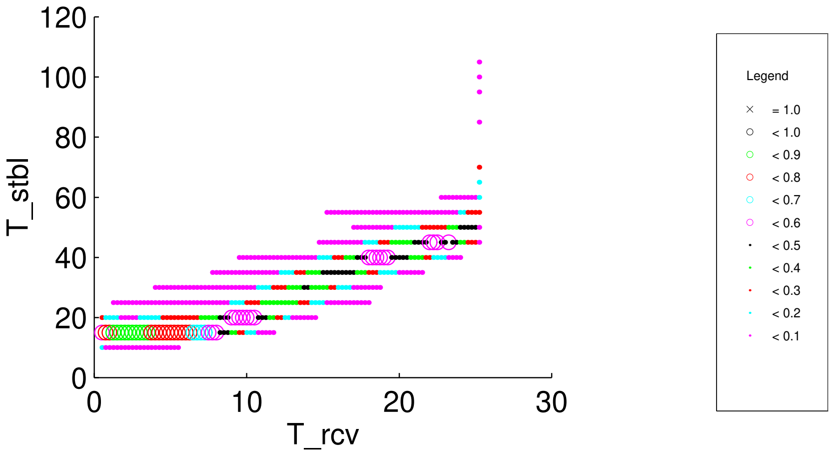


Figure 4.8: Sample probability distribution of T_{stbl} for a run of the process-based simulation.

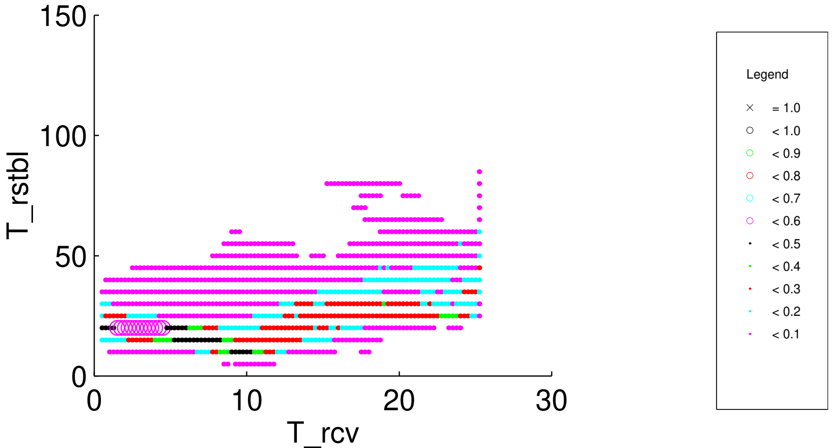


Figure 4.9: Sample probability distribution of T_{rstbl} for a run of the process-based simulation.

hardware parameters would be of interest because they may more clearly identify the type of hardware that limits or enhances the use of ordered multicast software.

Secondly, a number of parameters were identified as definitely having influence on the performance of the protocols. The parameters that fit into this category are either obvious candidates or are the result of the observations made in Chapter 3. These parameters include those that define the load on the network, process sending and receiving frequencies (μ_s and μ_r), group size (g), and degree of network stability desired (s). μ_s, μ_r, and g all influence T_{rcv}, T_{rstbl} and T_{stbl}, while s influences only T_{rstbl} and T_{stbl}.

The third and fourths steps were the development of two experimental designs. The first investigated the influence of the timeouts used in the software (t_a, t_{rs}, t_{ss}). The second investigated the influence of send window sizes (w_s, w_p). In each case, the influence of the group overlap parameters O and I was studied. Two separate experimental designs were used because the interrelationships between the two logical groups of parameters was assumed, for simplicity, to be non-existent. If there is a relationship this may mean that two parameters are identified as important when only one needs to be so identified. The high and low values used in each test are listed in Table 4.1. A complete list of the tests and their results

Table 4.1: High and low values for the parameters used in the first and second experimental designs.

parameter	high value	low value
O	5	1
I	5	1
t_a	1 s	10 μs
t_{rs}	.5 s	.1 μs
t_{ss}	.5 s	.1 μs

parameter	high value	low value
O	5	1
I	5	1
w_s	512	64
w_p	512	64

(as graphs) can be found in Appendix A. Each experiment was executed for four different values of μ_s. Hence each page of figures in the appendix contains four sets (rows) of graphs, one for each value of μ_s.

The figures in the appendix were inspected to identify which parameters seemed to have the greatest affect on the probability distributions of T_{rcv}, T_{rstbl}, and T_{stbl}. For example, consider the influence of the acknowledgement send timeout, t_a. Graphs in which all other parameters have been held constant except t_a are the pairs: A.2 - A.6, A.3 - A.7, A.4 - A.8, A.5 - A.9, A.10 - A.14, A.11 - A.15, A.12 - A.16, A.13 - A.17, A.18 - A.22, A.19 - A.23, A.20 - A.24, A.21 - A.25, A.26 - A.30, A.27 - A.31, A.28 - A.32, and A.29 - A.33. In each case the first figure in each pair is for the smaller value of t_a and the second is for the larger value. If we consider only the first column of graphs those for T_{rcv}, we see that while the graphs are not identical, there is no clear affect that results from the change in t_a. In general, the shape of the graphs remains similar when the value for t_a is increased. They differ in that the probabilities tend to be slightly smaller in some cases. Consequently, there is a greater probability that more of the times are greater than the maximum time recorded. We conclude, therefore, that T_{rcv} is weakly dependent on t_a. On the other hand, if we consider the graphs for both T_{rstbl} and T_{stbl}, we conclude that with the exception of the pairs A.3 - A.7 and A.5 - A.9 the delay clearly increases when t_a increases. Even in these two exceptions, the typical delay is likely to be greater — the distribution is less spread out in the second graph. We therefore conclude that T_{rstbl} and T_{stbl} are strongly dependent on t_a.

A similar analysis was done for each of the other parameters. The conclusions of this analysis (only strong relationships are recorded), combined with the observations of Chapter 3, are summarized as follows:

$$\begin{aligned} T_{rcv} &= f(u, \mu_s, t_{rs}, g, I, \mu_r) \\ T_{rstbl} &= f(u, \mu_s, t_a, t_{rs}, w_s, g, s, O, I, \mu_r \mid T_{rcv}) \\ T_{stbl} &= f(u, \mu_s, t_a, t_{rs}, w_s, g, s, O, I, \mu_r \mid T_{rcv}) \end{aligned}$$

4.5.3 Discussion

The experimental design shows that nine of the parameters studied have a noticeable effect on the behavior of a system. Unfortunately, using nine parameters results in a nine-dimensional space of probability tables. Even if computed for a few values of each variable, this space will be too large to compute easily. Therefore, it will be necessary to further limit the space by fixing more of the parameters for the remaining experiments. Of the nine parameters, μ_s, μ_r, g, s, O, and I are left variable. These parameters are selected because they allow us to study the effect of varying loads (μ_s, μ_r, and s), groups sizes (g) and organizational structures (O and I).

The resulting random cost functions are defined by a set of discrete input values. However, two of the input parameters, μ_s and μ_r, are continuous in nature. To use the probability tables, either the run-time values for these parameters must be rounded to the nearest possible value or some form of extrapolation needs to be used. We use a combination of both methods. Three nearby discrete pairs of values are identified. If one of these is sufficiently close (within some euclidean distance δ) to the continuous pair, the results of this pair is used directly. If not, we use the three points to define a plane through them. We then use linear extrapolation to approximate values for T_{rcv}, T_{rstbl}, and T_{stbl}.

4.6 Initial Experiences: Chains of Process Groups

Evaluation of the new simulation model was done by comparing the output of several simple tests with those obtained by the original process-based model. This comparison shows that, while the two models do differ, the new model does approximate the original model. The existence of differences between the models suggests that there are interactions not captured by the random functions. Investigation of these interactions reveals the interesting result that the delays imposed by the ordering protocols cause a "convoy" effect [BGMP79]: packets are accumulated at each point of delay into ever larger groups. The consequence is larger than expected delays due to an increase in network access times.

The simple experiments are now described and their results discussed in further detail.

4.6.1 Description of Experiment

A comparison between simulation models was done using a linear chain of process groups. Packets, originating at a single process, flow through the chain of processes.

They are forwarded from group to group by processes in the intersection of groups. For example, a five group chain is depicted in Figure 4.10. A chain of process

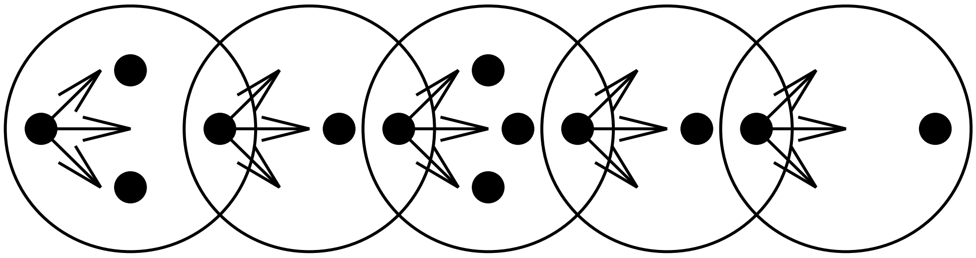


Figure 4.10: Linear chain of process groups. Packets are forwarded from one group to the next by the processes in the intersection of two groups.

groups was used because it is both simple to work with and it can be considered representative of a hierarchy of groups. A mapping can be made between a chain of processes and a path from the root of a hierarchy to one of its leaves.

In our experiments, chains of lengths up to 23 3-, 5-, and 7-member groups were built. Packets were forwarded from the first group at frequencies of 100, 200, and 300 packets per second. Measurements of the protocol delays were made at each link of the chain; ie, at each process in the intersection of two groups.

4.6.2 Results and Discussion

The results of these experiments show that the time required for a packet to be transmitted through a chain is slightly super-linear for the process-based model and linear for the group-based model. That is, in the process-based model the cost per link increases as a function of how far the link is from the packet source. The source of this increased cost was identified as increased delays for network access. These increased delays are the result of larger and larger clusters of packets being forwarded (hence passed to the network) at the same time. This fact is revealed in the histograms in Figure 4.11. They show the number of packets forwarded to the network in $1\mu s$ intervals at the originating process and at three other links in the chain. The number of packets being forwarded in a $1\mu s$ interval increases as the distance from the originating process increases.

This observation raised the following question: why does the number of packets forwarded at (approximately) the same time increase with the distance from the packet source? At the packet source, packets are uniformly distributed. Further, they are never delayed at the source due to the causal order protocols (they may, however, be delayed by full send buffers). However, at the first link, most packets are delayed both at reception and prior to send because the intersecting process is sending to one group as soon as it receives from another. Therefore a collection of packets accumulates at the first link: those packets which arrive while the process is halted awaiting stability information. This collection of packets can be forwarded at the same time to the next link. This process repeats itself at each link resulting in ever larger groups of packets being forwarded at nearly the same time. The histograms in Figure 4.11 show this phenomenon: the number of packets forwarded in a very short interval of time increases with distance from the source process. These histograms reveal this same fact in a second way: there are larger periods of time in which no packets are being sent. This occurs because packets are being sent in larger batches.

While the source of the added cost has been identified as increased network access time, it is a direct consequence of the delays imposed by the ordering protocol. This suggests that the ordering protocol imposes a limit on the throughput of systems that rely on them.

4.7 Summary

A simplification of the process-based model was described. It eliminates many, if not most, of the processes in a system — those which are members of only a single group in the system. Further, it employs experimentally determined random cost functions instead of simulating many costs. Both simplifications simplify the model and therefore reduce simulation costs.

The random cost functions are discrete functions defined by a discrete set of probability tables developed from a fixed set of input values. The desirable input parameters to the cost functions were determined by means of an experimental design that identified the most important parameters. Unfortunately, the design showed that many more parameters have an important effect than was hoped

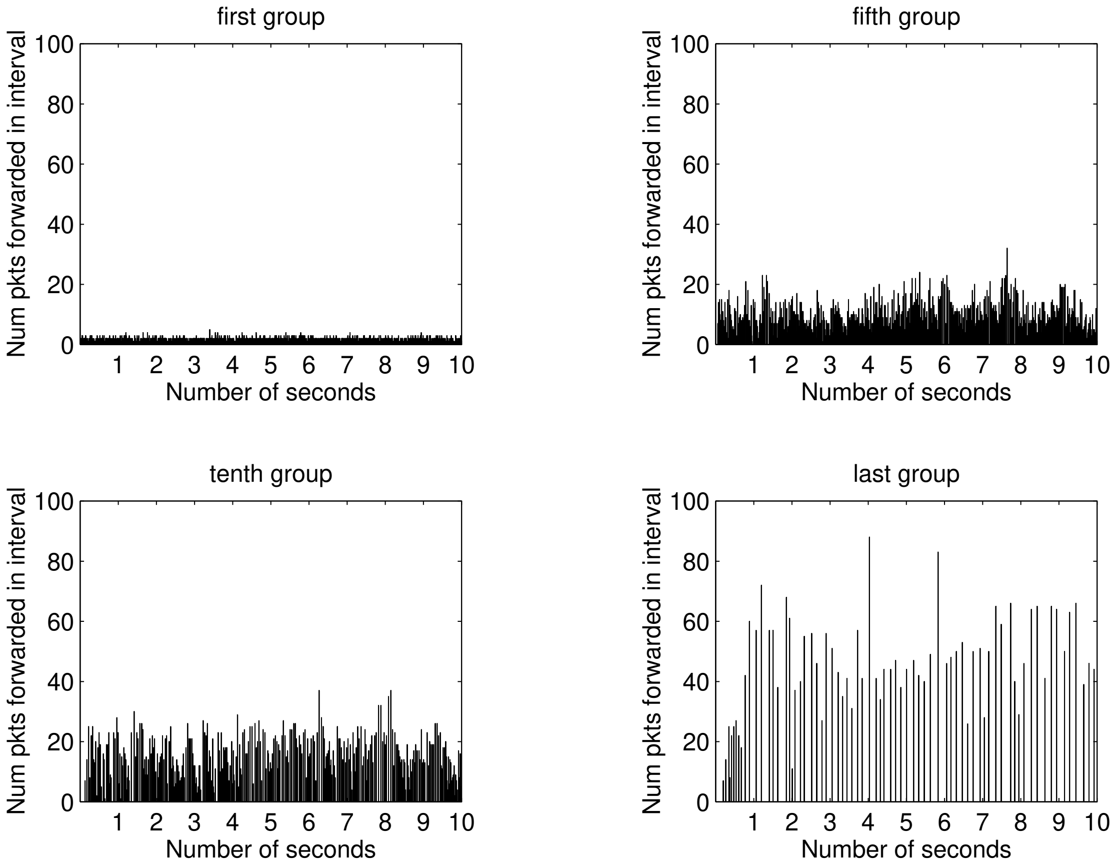


Figure 4.11: Number of packets forwarded in a 1 μs interval (over 10 s) in the 1^{st}, 5^{th}, 10^{th}, and 15^{th} groups of a chain of 15 5-member process groups.

for. Because of this, the number of probability tables needed for the random cost functions is very high. The cost to set them up becomes prohibitive for complex group topologies. However, they are relatively straightforward to develop for simple topologies such as chains, hierarchies, and peer groups. A second drawback to the cost functions is their use of extrapolation. The cost of this extrapolation replaces much of the savings made by using random functions. Therefore, this model, while more efficient, is not significantly more so.

Finally, the observation that group based systems lead to "convoys" was made when comparing the two simulation models. That is, it was observed that protocol delays lead to a grouping of packets into convoys. These convoys grow in size at each new delay. In turn, larger convoys result in larger delays and greater system burstiness as they monopolize shared resources like network bandwidth. Such convoys impose a limit on the size of a group based system in the sense that if they grow too large system resources are devoted to handling them. Convoys have been observed in other distributed settings such as databases [BGMP79]. In databases, convoys of transactions can occur when they are delayed waiting to be granted locks. Such convoys can be eliminated by removing the FIFO behavior of lock queues; that is, in modifying the algorithm used to grant the lock to another transaction [BGMP79,GR93]. Such an approach, however, does not seem to be readily applicable to this setting: FIFOness is inherent in a problem that requires ordered messages. The corresponding solution may be to structure the system to minimize the length of chains of process groups or to order packets only at the ends of such chains.

Chapter 5

Application: Information Services

5.1 Introduction

The preceding 2 chapters outlined approaches to simulating group based distributed systems. We first described a model that simulated the behavior of each process in the system and every packet transmitted by the system. We then refined this approach to eliminate "unimportant" processes and packets. As a consequence of these refinements, we are in a better position to study the behavior of causally ordered multicast in very large systems. Doing so may determine the usefulness of the process group paradigm for general distributed systems development by establishing limits on how large a distributed system requiring event ordering may be.

In this chapter we study large hierarchically structured systems. Such systems are being used for a number of *information services* such as information dissemination systems, such as publish/subscribe systems [Gla95] and directory services [CCI92,Vay95]. In an information dissemination system, a few processes are information sources. They "publish" information which is forwarded to the remaining processes which are information sinks, or "subscribers". For example, a stock market service may publish information about stock price changes to customers (subscribers) both on the trading floor and in traders' offices around the world. In such a system, because the number of subscribers is very large compared to the number of publishers, an efficient way to disseminate this information is to construct an hierarchy of process groups[1].

Large directory services, such as one defined by X.500 [CCI92], may contain widely distributed information, logically and/or geographically. This information is usually hierarchically structured. Clients, at the leaf nodes, submit queries to servers, non-leaf nodes, which may pass them up (or down) the directory hierarchy until they can be satisfied. Responses follow the reverse path. To ensure reliability (through replication), hierarchically structured groups of servers may be used to implement the directory service [Vay95].

This chapter seeks to identify limits of hierarchically organized systems. Possibilities that might be investigated are the size of the groups used to build the hierarchy, the number of data sources, the method of data forwarding used by intermediate nodes, and the height of the hierarchy. The larger the size of the groups, the longer it takes for messages to be received by all the members of the group. Because software multicast is used, the throughput decreases as the size of the group increases. A single group should not become a bottleneck for the whole system. Secondly, as the number of data sources increases, the number of potentially interfering messages increases. Multiple sources result in bidirectional traffic; that is, both up and down the hierarchy at the same time. This traffic may initially reduce the delays because acknowledgements are sent sooner (timeouts will not be necessary), thereby increasing throughput. However, eventually, the bidirectional traffic will result in more delays because of the ordering protocol: forwards will increasingly be asked to send in both directions simultaneously. Thirdly, data may be forwarded in a number of different ways by processes in the intersection of groups. One process can be (primarily) responsible for this task or one a number of processes can be responsible; either several to one child group or one for each child group. Finally, the convoy effect observed in Chapter 4 suggest that delays caused by the height of the hierarchy may be a factor limiting the practical height of the hierarchies. However, this effect is not fully captured by the simulation model and so will not be investigated fully here.

In order to study these issues, this chapter specifies a particular hierarchy of

[1]Note that a publisher need not be at the root of such an hierarchy; in general a publisher may also be a subscriber of some other publisher.

groups. A series of simulation experiments are defined and the results of their execution are presented. Conclusions from these results can then be drawn.

5.2 Experiments and Results

Experiments were designed to investigate the impact of three central structuring issues in hierarchical systems. These are the size of groups, the method of packet propagation (forwarding) used, and the effect of multiple sources. Our results indicate that relatively small groups should be used except as root nodes, that the task of forwarding packets from one level of the hierarchy to the next should be as distributed as possible among the members of a group, and that, except for an increase in network contention, multiple sources have little effect on the delays in hierarchical systems. Each issue is now briefly discussed. The experiments and their results are then presented.

As group size increases, the potential throughput of the system decreases because of network contention — software multicast is used in these experiments. Further, increased network contention may result in longer delays for the protocol: the time to transmit acknowledgements and stability information will increase. We find, in Section 5.2.2, that the performance degrades slowly with increased group size. Because the protocols are being executed primarily by processes in the intersection of groups, this suggests that smaller groups should be used in the interior of the hierarchy. Large groups can be used at the leaf level.

A second structuring issue is how data is forwarded from one group to another in the hierarchy. This task can be delegated to a single process or can be distributed among several process. In the second approach, a child group can be the responsibility of one or more process. We experiment with all three strategies in Section 5.2.3 and find that the task should be shared among as many processes as possible. In particular, at least one process should be assigned to a child group. The gain from more than one process assigned to a group is not large.

The final structuring issue investigated is the effect of multiple sources on the protocol delays and throughput. Most information applications will have multiple information sources and/or multiple clients (at the leaf level) making requests for information. Data will therefore “flow” in both directions through forwarding processes. Our experiments study how given flows in one direction influence those in the other direction. We find that throughput is lowered because of increased network contention. However, beyond this, the effect appears to be neutral: delays are little changed.

5.2.1 General Comments on the Experiments

Before presenting the experiments and their results, a number of general comments about all the experiments are made. These comments address the systems studied, the simulation parameters assumed, the type of experiments run, and the nature of the resulting figures.

The issues investigated result in very local results. That is, the figures presented below consider the costs at one or two processes within a large system. These numbers are independent of the height of the hierarchy. Thus most of the results can be obtained using relatively short hierarchies. Further, relatively small groups are used. This is done because the experiments done on groups size in Chapter 3 and below suggest that smaller groups are better — delays are lower and throughput is higher.

The experiments were done using a simulation of a high speed ATM network [dP93a,HHS94]. This simulation is only approximate in that it uses a high speed shared medium for the members of a single group. Other groups use other shared media. One way to view this is that each group shares a single virtual circuit, rather than each pair of processes sharing some number of virtual circuits (where this number depends on the number of ATM switches). This approach to simulating an ATM network, while not strictly accurate, eliminates the affect of communication by processes in other groups from all measurements. In a large system, the geographic distribution of the network as a number of subnets would have this same effect.

In each experiment two packet forwarding methods are explored: *dependent* and *independent* forwarding. In an information system, information is forwarded from one group to another until it reaches its destinations; either information consumers or servers that are able to respond to the request. This would typically be done by forwarding packets when needed; that is, when they are received and identified as requiring further forwarding. We call this *dependent* forwarding since the forwarding is dependent on the receipt of the packet from the previous group. We consider *independent* forwarding to be when packets are forwarded randomly and independent of their receipt. In particular, this means that a packet can be forwarded before its receipt. Doing this allows us to understand to what degree the dependence is responsible for the large delays or whether they can be eliminated by sending randomly.

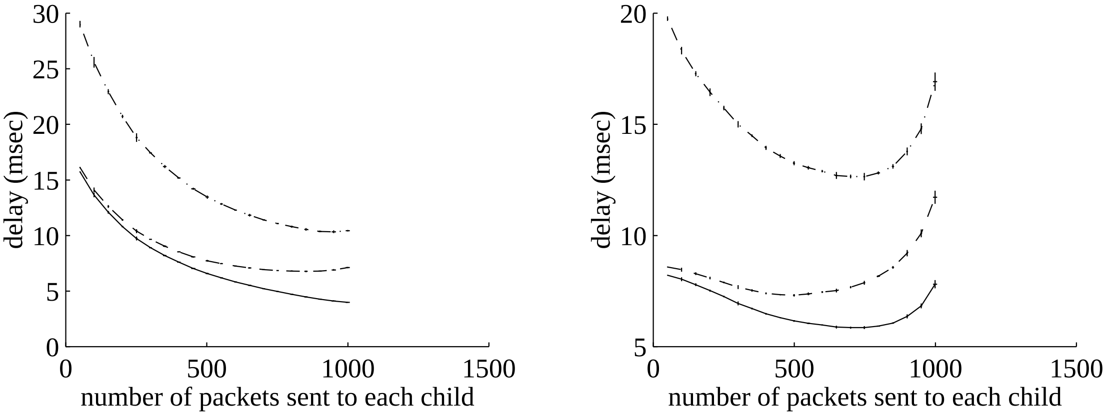
Finally, each experiment considers only the conservative causal multicast algorithm. Metrics measured include the cost of the delay prior to send, the cost to transmit a packet, and the cost of delay at receipt (prior to delivery). Combined, these quantities represent a worst case cost. The results of Chapter 3 show that for lower degrees of stability the results will be similar except in the case of 0- or 1-stability.

5.2.2 Influence of Group Size

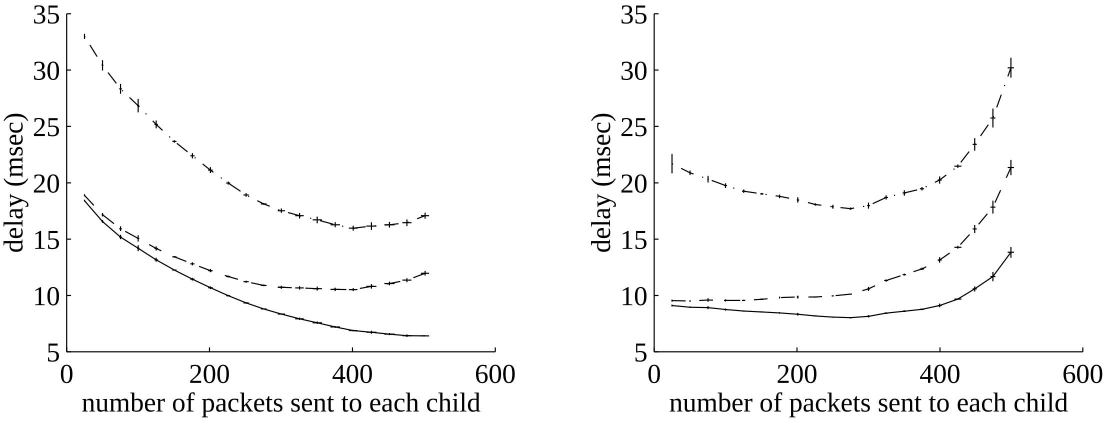
Experiments measuring the influence of group size are similar to those done in Chapter 3. They differ primarily in that those done here use the modified network simulation. This change makes delays independent of the height of the hierarchy. The delays presented in Chapter 3 were dependent on the height of the hierarchy because the network was shared by all processes in the hierarchy. Here, a network is shared only by members of a single group. Therefore, the load on it depends only on the load applied by the members of a single group.

Figure 5.1 shows the average delays for 5-, 9-, and 13-member groups. Both dependent and independent forwarding are shown. The graphs show that when throughput is low, all of the average delays increase only slightly as the size of the group increases. (The graphs in Figure 5.1 do not show this clearly since they use different axes. Figure 5.2 shows this more clearly by overlaying the 3 rows graphs in 5.1.) However, the maximum throughput of the group is much lower for larger groups. The lower throughput is caused by the use of point-to-point multicast instead of any hardware multicast. The resultant total throughput is limited by the effective bandwidth of the network.

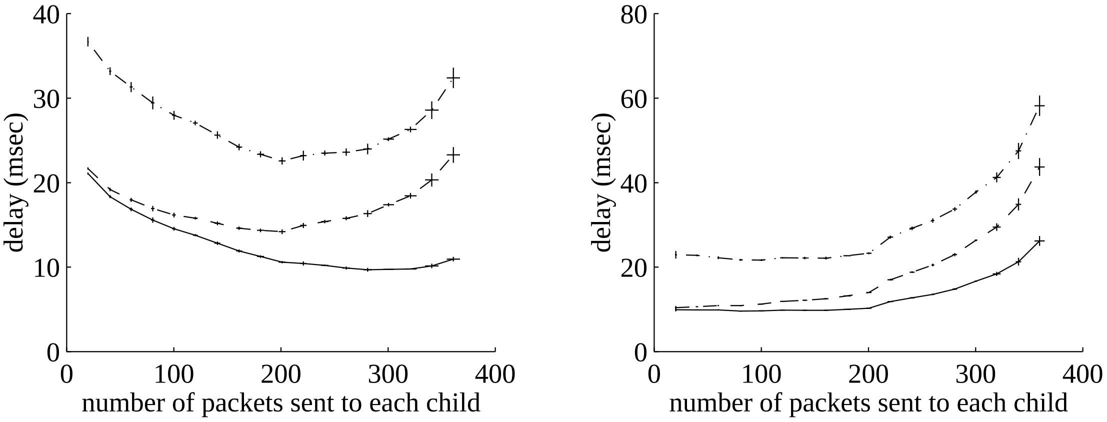
Note that the average delays decrease as the number of packets forwarded



(a) 5-member group; dependent and independent forwarding



(b) 9-member group; dependent and independent forwarding



(c) 13-member group; dependent and independent forwarding

Figure 5.1: The average delays at send, for transmission, and at reception for 5-, 9-, and 13-member groups.

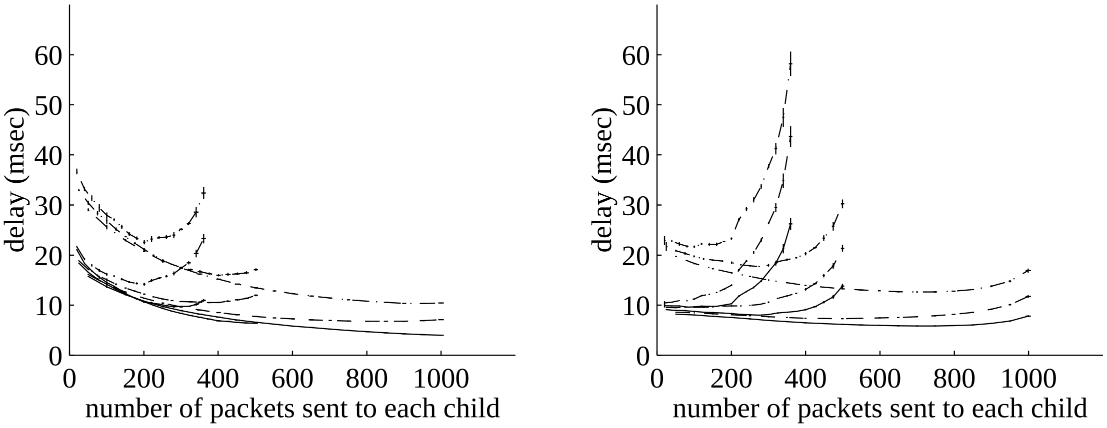


Figure 5.2: The average delays from Figure 5.1 overlaid for comparison. The first graph is for dependent forwarding, the second for independent forwarding.

increases until some limit at which point the average delays rapidly increase. This is caused by an increase in return traffic; that is, traffic from the destination processes. By default, data packets are sent only from the root group to the leaf groups. As the applied load (number of sends) is increased, there is an increase in the amount of acknowledgement information returned. This increase results from both an increased number of requests for stability information and from a greater number of full receive windows (which trigger acknowledgements in advance of the usual timeout). The quicker acknowledgement by destinations results in shorter delays by the senders, including the forwarder. Independent forwarding mitigates this effect. This is because use of independent forwarding already results in shorter delays. The delays are shorter because some packets have already become stable when requests for information are made and because attempts to send may not interfere as much with a receipt of an unstable packet. Therefore delays are either not necessary or are shorter. A consequence is that less return traffic needs to be solicited and the information provided is less valuable.

5.2.3 Different Approaches to Forwarding

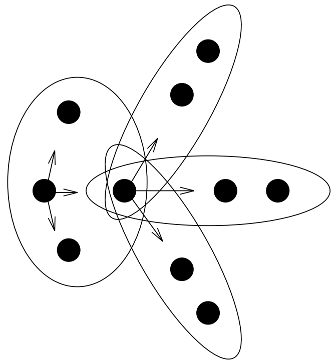
There are a number of different strategies to structure the forwarding processes in an hierarchical system. One process might act as a forwarder for a number of children groups, separate processes might be responsible for a single child group, or a number of cooperating processes might be responsible for a single child process. These different strategies are shown in Figure 5.3. Because delays are caused by attempting to send a packet after having just sent/received one from another group it would be expected that the performance would be worst for the first case and best for the last case. Figures 5.4, 5.5, and 5.6 show the results of these experiments.

In the first experiment (Figure 5.4), one process forwards packets to all the child groups. That is, one process is receiving and immediately attempting to send to a number of different groups at the same time. As a consequence, each packet is delayed before being forwarded to each group. We might expect this delay to be cumulative: that the delay for 2 children is twice that for 1, for three is three times as great and for 4 is four times as great. The graphs in Figure 5.4 do not reflect this because they represent average delays; averages over all the child groups. They do, however, show that there is an increase in delay as the number of children increases. As in all the other figures, the average delay decreases as the number of packets forwarded increases until some limit at which point the average delays rapidly increase.

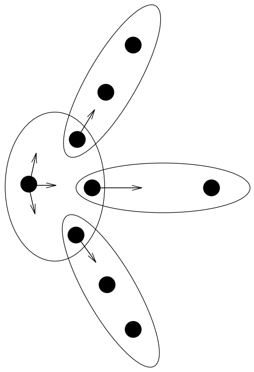
Finally, note that the number of packets forwarded to each child is inversely proportional to the number of child groups present. The effective maximum throughput of the forwarder remains constant. However, since it is forwarding to more groups, the effective throughput of the system drops as the number of children increases.

Figure 5.5 shows the average delays incurred when each child group receives packets from a single forwarding process in the parent group. That is, a forwarding process is responsible for forwarding only to a single child group. In this case, we would expect that the delays incurred by each forwarder would be approximately the same, independent of the number of child groups. Further, we would expect the maximum number of packets forwarded to each child group to remain constant as new child groups are added to the system. Figure 5.5 does, indeed show these results: the average cost per send is independent of the number of child groups and the maximum number of packets forwarded to each child is approximately 1400 (per second) before the average delays increase due to network contention.

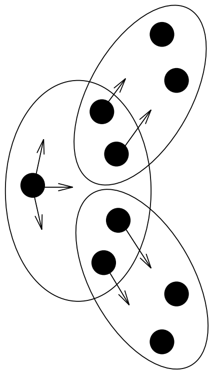
The final strategy for forwarding packets is to distribute the task of forward-



(a) one process forwards to all child groups

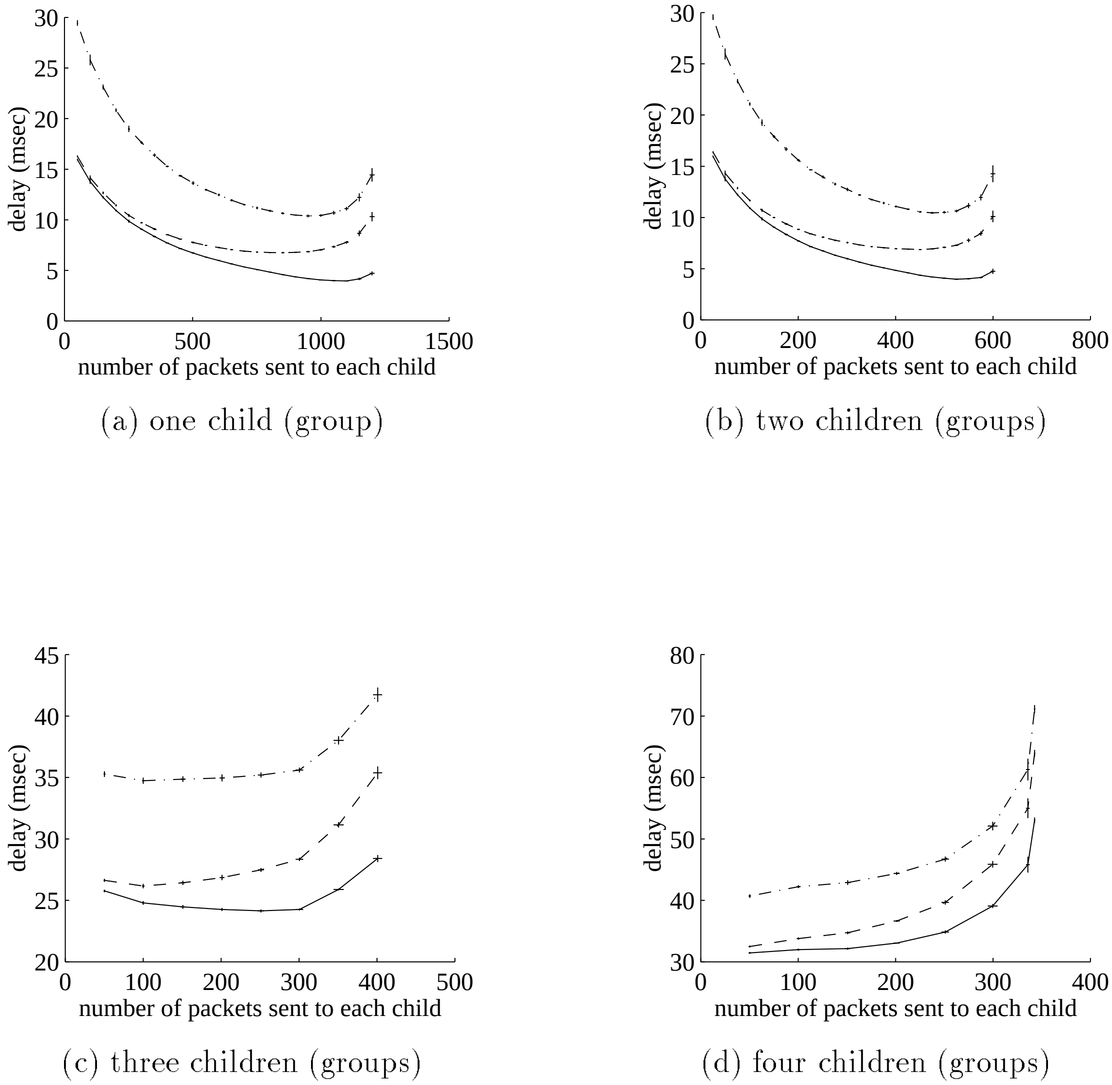


(b) one process forwards to each child group



(c) a number of processes forward to each child group

Figure 5.3: Different structuring strategies for forwarding packets in an hierarchical setting.



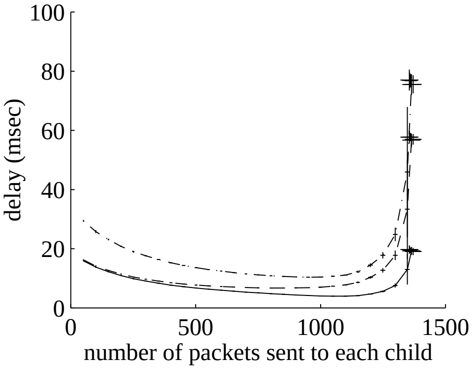
(a) one child (group)

(b) two children (groups)

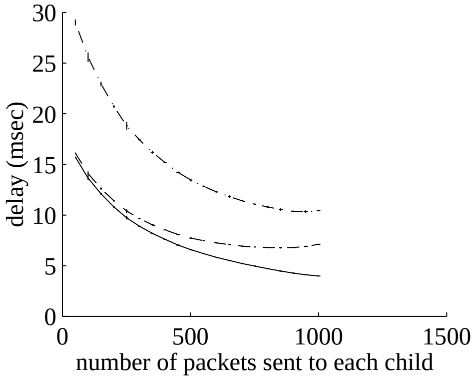
(c) three children (groups)

(d) four children (groups)

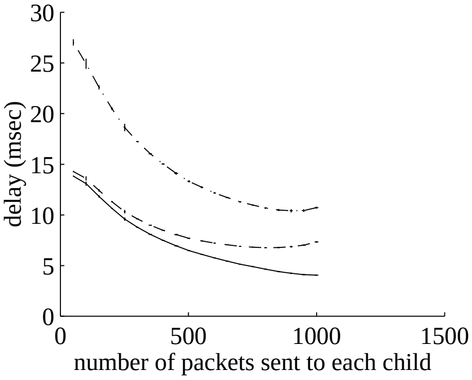
Figure 5.4: Cost of dependent forwarding in the case where one process forwards packets to all the children (groups).



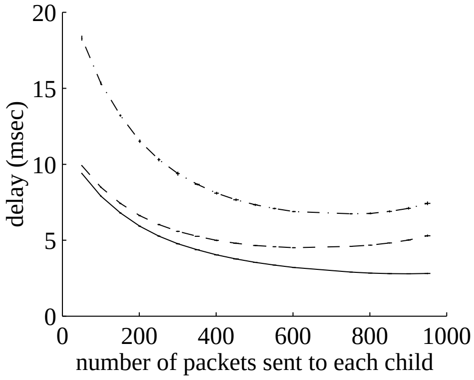
(a) one child (group)



(b) two children (groups)



(c) three children (groups)

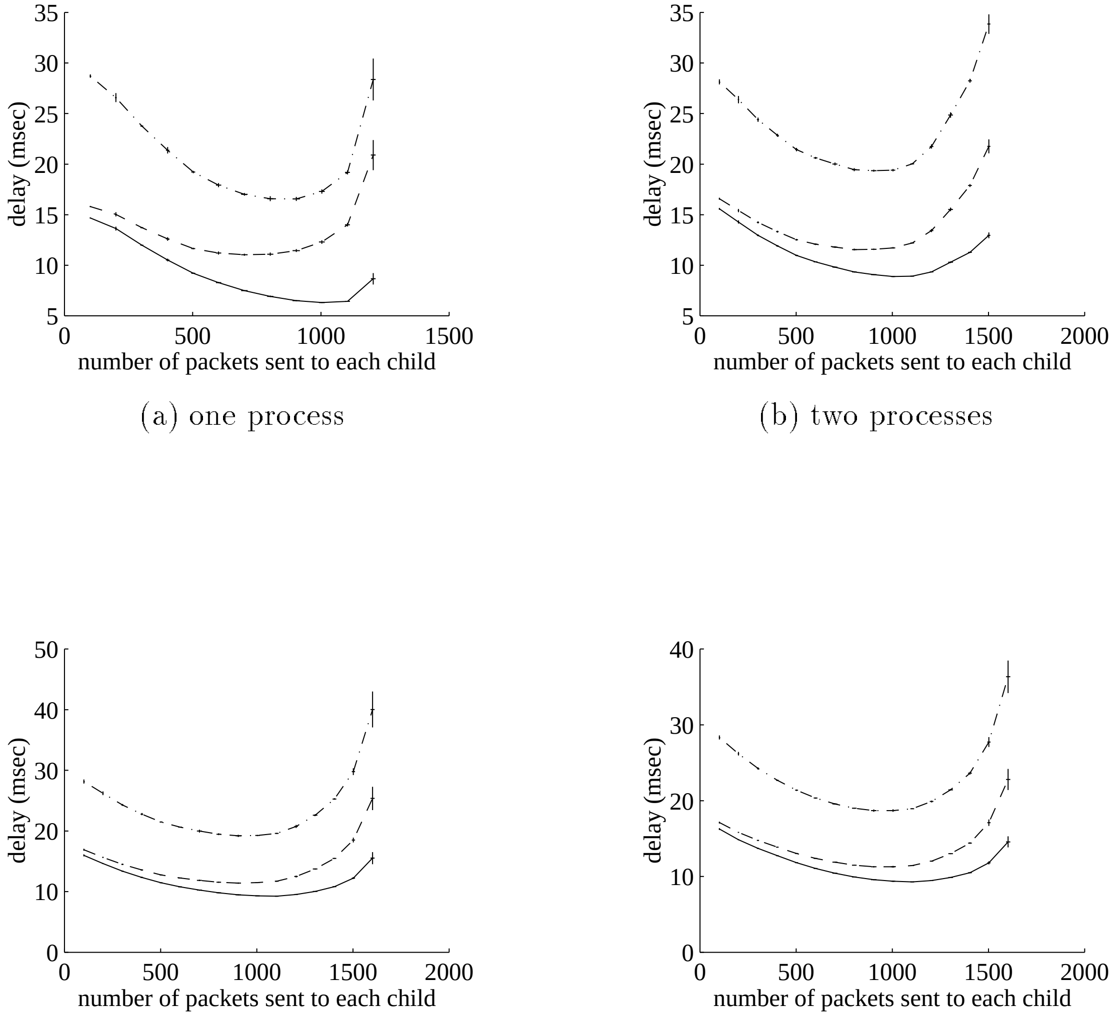


(d) four children (groups)

Figure 5.5: Cost of dependent forwarding in the case where one process forwards packets to only one child (group).

ing to a single child group among a number of parent processes. We experiment with between 1 and 4 such processes for each child group. Unlike the above two strategies, the number of child groups remains fixed at two (in the above two experiments, it varies from 1 to 4). Therefore, for purposes of comparison, the graphs in Figure 5.6 are comparable only to graphs 5.4(b) and 5.5(b). Further, in order for each child group to be "serviced" by up to four parent processes, 9-member groups were used instead of 5-member groups. This makes the results incomparable to either of the previous experiments. However, the general trends can be compared. In this final set of experiments, the average delays incurred by the forwarders is independent of the number of forwarders. It might be expected that because each forwarder has less work to do, delays would be smaller. However, the use of timeouts in the hopes of piggybacking information masks any gain that might come from having multiple forwarders — each must delay approximately the same amount of time for acknowledgements and stability information. Finally, note that the throughput of the system is also independent of the number of forwarders. It might be expected that the throughput of the system would increase with more forwarders — the work of forwarding is distributed rather than centralized. This reflects a limit on the original sender, not on the forwarders. These results suggest that distributing forwarding does not yield much of a gain unless there are multiple originators as well.

The results of the experiments for each of the three forwarding strategies reported above considered only dependent forwarding. That is, packets were forwarded immediately upon their reception. Each of the experiments was repeated using independent forwarding; packets were forwarded independent of their reception. It might be expected that the random nature of independent forwarding would result in lower delays because a packet might be forwarded some time after its reception, perhaps long enough after for stability information to have already propagated throughout the system. Figures 5.7, 5.8, and 5.9 show the results of these repeated experiments. They show that for low loads there is a lower average cost. For high loads the costs are about the same.



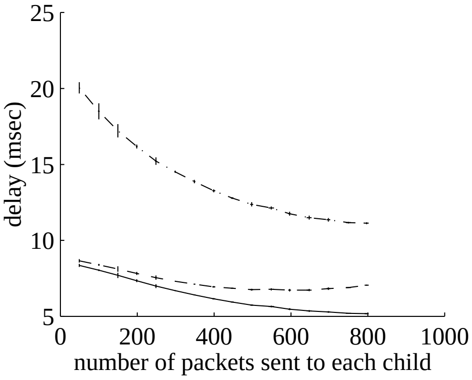
(a) one process

(b) two processes

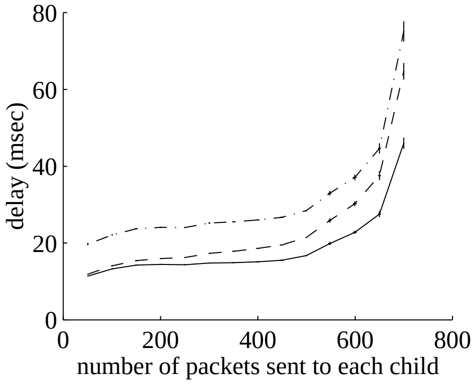
(c) three processes

(d) four processes

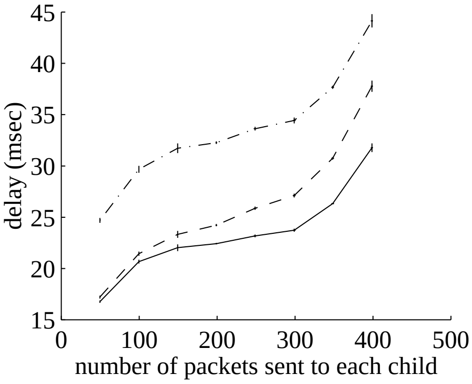
Figure 5.6: Cost of dependent forwarding in the case where several processes forward packets to one child (group).



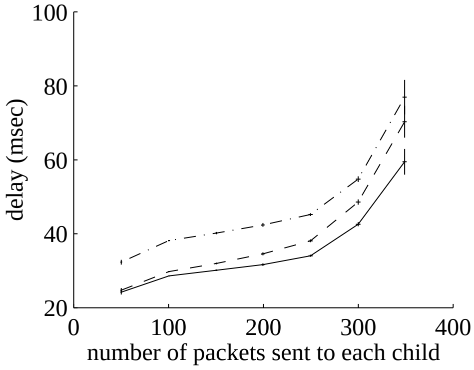
(a) one child (group)



(b) two children (groups)

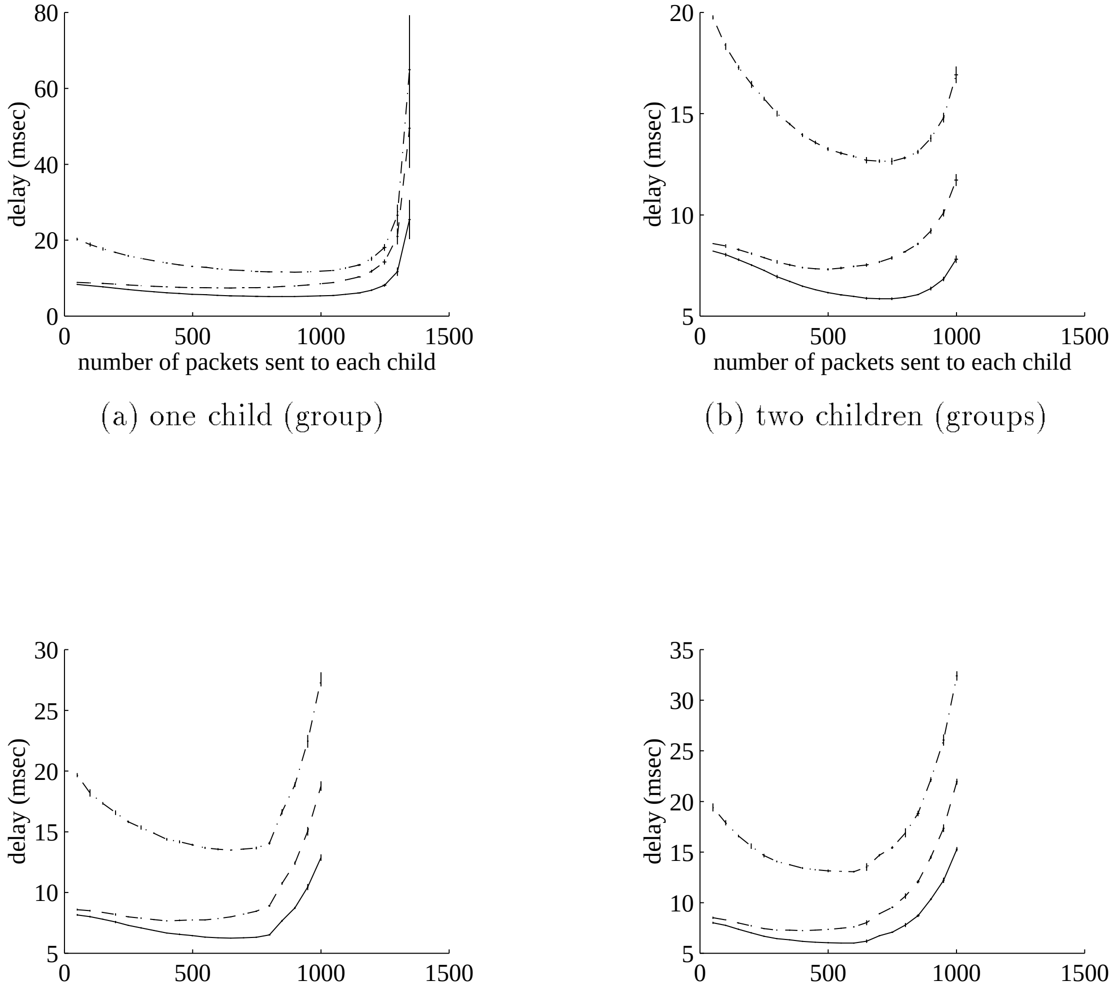


(c) three children (groups)



(d) four children (groups)

Figure 5.7: Cost of independent forwarding in the case where one process forwards packets to all the children (groups).



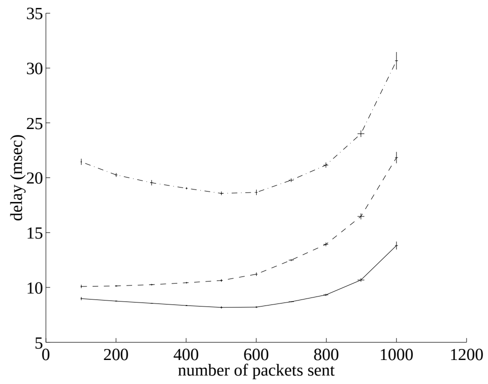
(a) one child (group)

(b) two children (groups)

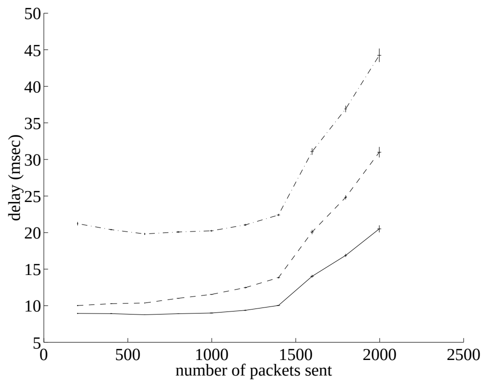
(c) three children (groups)

(d) four children (groups)

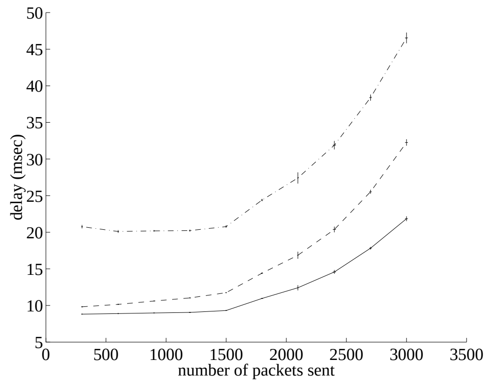
Figure 5.8: Cost of independent forwarding in the case where one process forwards packets to only one child (group).



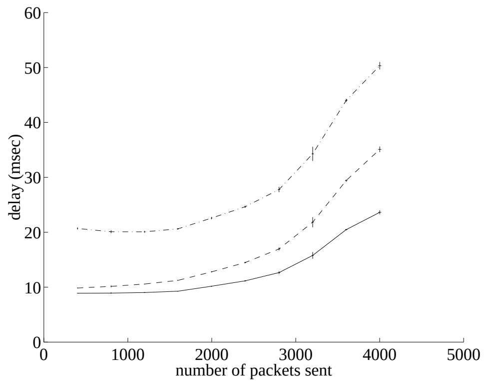
(a) one process



(b) two processes



(c) three processes



(d) four processes

Figure 5.9: Cost of independent forwarding in the case where several processes forward packets to one child (group).

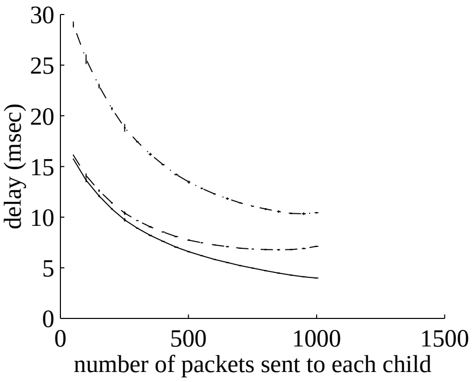
5.2.4 Multiple Sources

An information dissemination system is unlikely to have a single source of information at the root of the hierarchy (as has been the case in the experiments presented thus far). It is more likely that a number of sources is distributed throughout the hierarchy. A well structured system might try to place the most prolific sources at, or near, the root so that the path length, and hence overall delay, for most information is minimized. A directory service will have a number of clients at the root level and many information providers (servers) throughout the interior of the hierarchy. Many will be simultaneously sending requests and responses through the hierarchy.

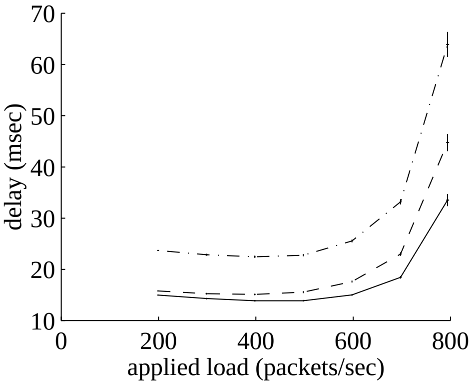
We now look at the effect of placing multiple sources in an hierarchy. We study the behavior of a single forwarding process when it is receiving varying amounts of information from both sources. In particular, we look at the effect of two sources. One of the sources will be the "primary" source. We measure the performance of sends from this source through a forwarding process when that process experiences varying loads imposed by some "secondary" source. We find that a secondary source lowers the delays of a low throughput primary. However, its effect is negligible as the load imposed by the primary increases. Finally, a secondary reduces the maximum throughput of the primary because of increased demands on the network resources.

The experimental situation considered is identical to the case depicted in Figure 5.3(b) with 2 children. Therefore, when the secondary sender sends 0 packets per second, the results are identical to those in Figure 5.5(b).

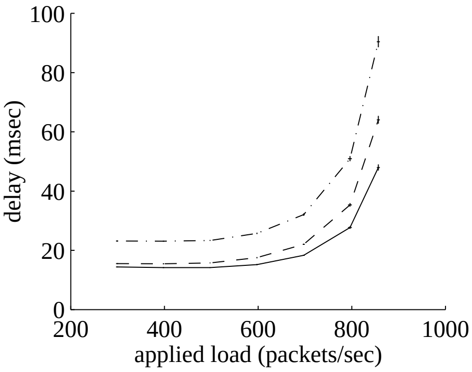
Figure 5.10 depicts the average delays for sending from a primary when the secondary is sending 0, 100, 200, and 300 sends per second through the same forwarder. It shows that the additional packets result in lower average delays only for very low primary loads. When the primary sends at higher loads the average delays are longer. The lower delays result from the demand of the secondary for the forwarder to switch the direction of its sending. This demand includes acknowledgement information that reduces the needed delays. However, higher secondary sending frequencies do not provide any additional information. Therefore, the period over which higher frequencies cause lower delays is smaller than in earlier



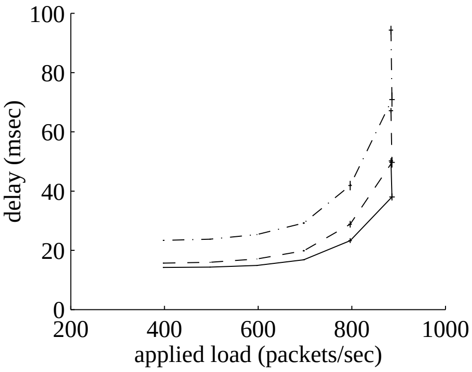
(a) 0 sends per second



(b) 100 sends per second



(c) 200 sends per second



(d) 300 sends per second

Figure 5.10: Cost of dependent forwarding in the case where one process forwards packets to each child (group).

experiments. Surprisingly, the sending frequency of the secondary sender has little effect on the average delays. That is, above some threshold value — less than 100 sends per second in our experiments — secondary sending frequency does not cause larger delays. However, the maximum throughput of the primary sender is reduced when the secondary is added. This is expected because of limited network bandwidth.

5.3 Discussion and Conclusions

A number of experiments looking at the effect of group size, different approaches to packet forwarding, and multiple senders were carried out. Our results show that smaller groups have lower average delay. Therefore smaller groups should be used in the interior of an hierarchy of groups. However, the size of the groups needs to be balanced with the observation that distributing the work of forwarding among many processes is advantageous. In particular, at least one process should be assigned to forward to each child group. If a single process is responsible for more than one child, the throughput of the system degrades as a function of the number of children served by this forwarder. Finally, our results showed that multiple senders reduce delays for low throughput primary senders. However, they increase the load on the forwarding processes which become bottlenecks. The resulting throughput of primary senders decreases.

All of the experiments were done by looking at the "microscopic" level. That is, measurements were made for a single forwarder within a larger system. At the "macroscopic" level, the delays observed will be repeated at each forwarder in the hierarchy. There are a number of consequences of this. First, if we assume in an information dissemination system that every node sees every packet, the throughput of the overall system is limited by the throughput of the slowest set of forwarders in the hierarchy. That is, because forwarders become bottlenecks, one pair of groups may limit the whole system. On the other hand, if we assume that only subtrees of an hierarchy see packets (for example, in a directory service), the throughput of the system may be higher. Second, the repetition of delays at each level combined with the development of convoys (see Chapter 4) means that the applied load per unit time becomes more bursty. It becomes characterized by large instantaneous loads followed by long periods of no load. It follows that delays at subsequent forwarders should be computed using instantaneous loads rather than average ones. As a consequence, our results indicate that the delays may initially decrease if there is only one sender in the system. With two or more senders, the delays will increase.

A possible solution, not investigated here, would be to remove most of the delays by limiting ordering to critical points (defined in some application specific way) or by relegating the responsibility of ordering messages to the end processes. In this way, delays would occur only at the end. It is not known if these delays will be greater than the cumulative delay. They may be very large in a large system because the number of potential conflicts will be higher.

Chapter 6

Discussion and Conclusions

In the previous chapters we described strategies to simulate the performance of causally ordered multicast protocols. The simulators developed were used to study the performance of both a number of small systems and aspects of a large information dissemination system.

We now consider the alternative approaches that could have been used to study the performance of ordered multicast: performance measurement and analytic modeling. We survey the results that have been obtained using these methods. Following these is a discussion of the ways in which our work could be extended. Finally, we conclude.

6.1 Alternative Approaches

Two alternatives to simulation for determining the cost of ordered multicast are performance measurement and analytic modeling. Both have been used to determine the cost of ordered multicast systems. We used simulation because it tends to more accurately model a system than does analytic modeling and because it allows for studies of larger systems than is possible with performance measurement. We briefly look at the relative merits of each approach and survey the studies of ordered broadcast that have been carried out using each approach (including simulation).

6.1.1 Analytic Modeling

Analytic modeling involves developing a mathematical model of a system. Solving the model for fixed values of the model parameters results in cost estimates. Because such systems can be solved for a wide variety of parameters, analytic models are a good way to identify asymptotic behavior. On the other hand, because group based systems are complex, a number of simplifications must be made to make the models solvable using numeric software. These simplifications may eliminate many significant sources of cost. Consequently, such models tend to be the most inaccurate.

Two studies, [May92,MMS94], model systems and estimate the cost of total order. [May92] assumes a fixed cost for each message delivery and computes overall cost by multiplying this fixed cost by the expected number of messages needed to guarantee total order. [MMS94] makes probabilistic assumptions about message reception and delivery. The resulting Markov model is then used to approximate delays.

Studies of causal order have only provided counts of the number of messages needed to guarantee order [BSS91,LLSG92]. In addition to estimating the number of messages needed to order messages, [BSS91] looks at the expected size of the vector clock that must be sent on each message and [CS93] estimates the growth in the size of the buffers needed by ordering protocols. Both of these studies use worst case assumptions.

6.1.2 Simulation

Simulation involves creating a model of a system and executing that model. Simulation models, like analytic models, usually involve simplifying the original system. However, the simplifications are not usually as great as those used in an analytic model. Indeed, the computational approach of simulation means that the amount of simplification is a model design decision that allows for an appropriate level of detail in the model. For this reason, simulations are generally more accurate than analytic models. Further, simulations are useful because they can be used to investigate either non-existent systems or systems larger than readily available resources.

The only prior simulation study of ordered multicast protocols is a study of the original ISIS protocols [BJ87b] in a single process group [DB91,Des91]. These protocols are significantly different (more inefficient) than those studied here. The results of this simulation match the expectations of a simple analytic similar to those in [May92].

6.1.3 Performance Measurement

Performance measurement involves instrumenting the actual code of a system and measuring the performance of the system as it executes under either real or test loads. Because the actual cost of a system is being measured, this method has the potential to produce the most accurate cost measures. However, it may suffer from an inability to accurately isolate the cost of particular pieces of a protocol from the costs of other system or external events. Further, this approach requires access to as large a system as is needed for the studies in question. For studies of scale, this implies access to a very large number of resources. Finally, while performance measurements from similar systems are often compared, such comparisons are usually invalid because they are frequently based on entirely different host architectures.

Traditionally, this is the method used to measure the performance of group based systems. For example, [KTHB89,ADKM92b,AMMS+93,WMK94,vHB94] present the cost of guaranteeing total order and [PBS89,BSS91,BC94] the cost of guaranteeing causal order. Each of these studies considers only single groups. They are incomparable in the sense that each uses different hardware. Further, [KTHB89] is a kernel level construct and hence has lower costs than the other, application level, systems. [ADKM92b,BC94,WMK94,vHB94] use hardware multicast facilities while the others do not.

6.2 Simulation Extensions

The work presented in this dissertation can be extended in two directions. The first direction is to use the simulation to carry out a number of additional studies. For example, the following issues might be studied: the cost of guaranteeing orders other than causal, the effect of membership changes, the implications of different definitions of virtual synchrony, and the influence of new hardware and operating systems. Also, the simulation could be used as a tool to identify probablistic real-time guarantees for message delivery. The second direction for extending this work is in the area of simulator refinement. In addition to making the simulation more efficient, refinements might include methods to simplify the implementation of new protocols. Each of these extensions is now briefly discussed.

In addition to looking at causal ordering, stronger ordering protocols could be studied using simulation. Such orders might include totally ordered multicast (see Section 2.3.2), and safe multicast [ADKM92b].

Additional studies might look at the additional influence of changes to group membership. Typically, membership changes result in delays in sending to ensure that the system remains virtually synchronous. Studying this additional cost of membership change may suggest ways of either combining delays or reducing them. It would also allow for a comparative study of differing definitions of virtual synchrony such as [BJ87a,MAMSA94,Fv95]. Such a study could compare the membership protocols themselves and their effect on ordered multicast.

Another set of studies could be carried out to investigate the influence of the hardware on ordered multicast protocols. Newer hardware is rapidly reducing network delays. To take full advantage of this lower delay, innovative approaches to reducing the operating system cost of message passing are being investigated. Examples include [BALL91,DP93b,Ros95,BBVv95]. The effect of a reduced path may influence both the rate of out of order messages and the delay when messages are out of order. In particular, is the reduction proportional to the speedup in message delivery of new hardware? Our results indicate that this relative cost is most important because it can result in making the system more bursty. If the relative delays become greater, the amount of burstiness will increase.

Finally, the results of the simulation could be used as a basis for establishing real time guarantees for the delays. The approach would be similar to that used in [MMS94]. In this work, probabilistic bounds on the time to deliver totally ordered messages were determined from an analytic model. Using simulation, similar bounds could be derived for a causally ordered messages. Further, the assumed probability distributions in [MMS94] could be more easily modified in a simulation — they will not be limited to something mathematically tractable. Before such work could be done, it is likely that the network models used in our simulation would have be more sophisticated.

In addition to using the existing simulators for investigating different aspects of group systems, the simulator itself could be changed. Presently, implementing a particular protocol is not straightforward: the protocol implementer must worry not only about the protocol details but details of simulator state and about any assumed underlying communication system. Further, for reasons of efficiency, there is no clear separation between the different communication levels. As a result, changing the ordering protocols requires extracting the current one and inserting the new one from several places. Finally, the implementer must think in terms of simulation events rather than in terms of protocol events and in terms of the simulator structure instead of the protocol structure. A different approach would be to allow a higher level description of a protocol that could be automatically augmented to deal with everything else. Further, a cleaner separation between levels of the communication protocols may allow for still greater flexibility of the simulator.

6.3 Related Work

An approach to increasing the scalability of group based systems is to eliminate ordered messages. The semantics of many applications preclude this option. However, it is also true that not all messages need to be ordered. Indeed, application semantics may require ordering only at some critical points but not at intermediate points or may require ordering among several different classes of messages but no ordering between these classes. Each class may have different ordering requirements, or none at all. Further, in the context of overlapping groups, a number of different ordering semantics are possible [BCG91,GT90]. Applications may require different semantics in different overlappings.

There are two proposed approaches to reducing the number of messages that must be ordered. One approach is to offer *causality domains* [BCG91], similar to *conversations* in Psync [PBS89]. These are separate domains in which different ordering requirements would be guaranteed. The idea is that logical sets of process groups would be guaranteed whatever ordering requirements are needed. Different domains could be defined for different (and unrelated) classes of messages exchanged by the same set of process groups. This solution has not been implemented for multiple overlapping groups. The second approach, based on the end-to-end argument [SRC84], is to entirely remove ordering guarantees from the group subsystem. Guaranteeing order becomes the responsibility of each application program. The gossip architecture [LLSG92] uses this approach for application level communication. In this approach, an application requiring a message to be ordered, must explicitly attach the necessary vector clock to the message. Very large systems which are very likely to be partially disconnected at all times also use this approach. For example, RELACS [BDGB95] implements only group membership algorithms and provides virtual synchrony[1] without message ordering.

Both approaches may be used in a system which decouples the ordering semantics from group membership and virtual synchrony guarantees. Different groups can then be defined for different communication requirements. Each group[2] can then be provided with whatever guarantees are needed, included none (in which case the application program must provide them, if needed). This approach is used in Horus [vHB94,vB95] where a communications "stack" defines the communication protocol of a group. Several stacks, defined on the same group, can be used to provide different ordering requirements.

6.4 Conclusion

Two simulators were constructed to investigate the performance of causally ordered multicast in settings where there are large numbers of overlapping groups. The first simulator considered the effect of every process and packet sent on the overall system. It was used to do a number of small case studies that showed that the number of packets that arrive out of order can be small. However, the group structure — the way in which membership information is maintained and used can greatly influence the number of packets that arrive out of order. Further, when they do arrive out of order some packets are delayed a long time. The number of out of order packets can also be reduced by delaying packets before they are

[1] Termed *view-synchronous communication* in RELACS.

[2] Note that a group here is a logical notion: several groups can be mapped to the same set of actual processes. Each group defines a different set of communication guarantees.

sent. This approach has low cost when subsequent multicasts are made to the same group. (This is expected to be the common case; see, for example, [Mog92]). When a change occurs, the delays are very large: orders of magnitude higher than the cost to send without a delay.

The large delays of the conservative protocol seem to be highly dependent on the performance of the hardware (including kernel costs), the underlying method of flow control, the acknowledgement scheme, and the timer algorithms. New technologies and new algorithms (for example, U-Net [BBVv95], a user level network interface, especially for ATM networks) will reduce some of these delays. However, our results indicate that size of the relative delay (between sending with and without a delay at send) is more important. If this relative delay remains unchanged, our results show that the delays will still result in greater system burstiness. That is, packets filtered through a sequence of groups (ie, through a sequence of delays) will tend to become more bunched, or "convoyed" together. These convoys result in greater burstiness which, in turn, results in greater network contention and still larger convoys (ie, greater delays). An applications tolerance of this convoy effect defines the limits to its scalability. Indeed, it suggests that applications should be designed to avoid such a communication pattern whenever possible. This can be done by avoiding communication patterns that require ordering messages across many group boundaries, sending to the same group whenever possible, batching sends to new groups, and by not enforcing ordering requirements until it is necessary to do so. The related technique of causality domains, if sufficiently general, may provide the ability to implement such strategies in a system.

Appendix A

Appendix: Results of Experimental Design

The graphs on the following pages represent the results of the experimental design described in Section 4.5. The first 32 pages of graphs are from the first experimental design and the remaining 6 pages are from the second design. Each page of 12 graphs represents the output of a single experiment in the design. Within each row, the first graph represents the probability distribution of T_{rcv}. The second and third graphs represent contour maps of conditional probabilities $T_{stbl}|T_{rcv}$ and $T_{hstbl}|T_{rcv}$, respectively (The legend in Figure A.1 should be used to understand these contour maps). All axes are milliseconds except the y-axis in the first graph which represents probability. Tables A.1 and A.2 detail the parameters of each experiment in the design. Each experiment was done using four different loads. Hence each page contains 4 sets (rows) of graphs.

A more complete explanation of how to read the graphs can be found in Section 4.5.1 and their use is detailed in Section 4.5.2.

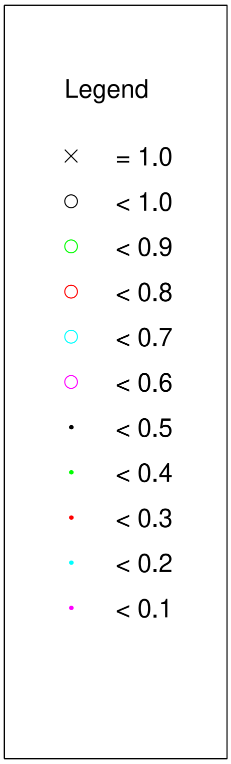


Figure A.1: Legend showing correspondence of symbols to probabilities in the conditional probability contour maps in this Appendix.

Table A.1: Input values for the first experimental design. The results are in Figures A.2 through A.33.

Figure	O	I	t_a	t_{rs}	t_{ss}	Figure	O	I	t_a	t_{rs}	t_{ss}
A.2	5	5	10 ms	.1 ms	.1 ms	A.6	5	5	1 s	.1 ms	.1 ms
A.3	5	5	10 ms	.1 ms	.5 s	A.7	5	5	1 s	.1 ms	.5 s
A.4	5	5	10 ms	.5 s	.1 ms	A.8	5	5	1 s	.5 s	.1 ms
A.5	5	5	10 ms	.5 s	.5 s	A.9	5	5	1 s	.5 s	.5 s
A.10	5	1	10 ms	.1 ms	.1 ms	A.14	5	1	1 s	.1 ms	.1 ms
A.11	5	1	10 ms	.1 ms	.5 s	A.15	5	1	1 s	.1 ms	.5 s
A.12	5	1	10 ms	.5 s	.1 ms	A.16	5	1	1 s	.5 s	.1 ms
A.13	5	1	10 ms	.5 s	.5 s	A.17	5	1	1 s	.5 s	.5 s
A.18	1	5	10 ms	.1 ms	.1 ms	A.22	1	5	1 s	.1 ms	.1 ms
A.19	1	5	10 ms	.1 ms	.5 s	A.23	1	5	1 s	.1 ms	.5 s
A.20	1	5	10 ms	.5 s	.1 ms	A.24	1	5	1 s	.5 s	.1 ms
A.21	1	5	10 ms	.5 s	.5 s	A.25	1	5	1 s	.5 s	.5 s
A.26	1	1	10 ms	.1 ms	.1 ms	A.30	1	1	1 s	.1 ms	.1 ms
A.27	1	1	10 ms	.1 ms	.5 s	A.31	1	1	1 s	.1 ms	.5 s
A.28	1	1	10 ms	.5 s	.1 ms	A.32	1	1	1 s	.5 s	.1 ms
A.29	1	1	10 ms	.5 s	.5 s	A.33	1	1	1 s	.5 s	.5 s

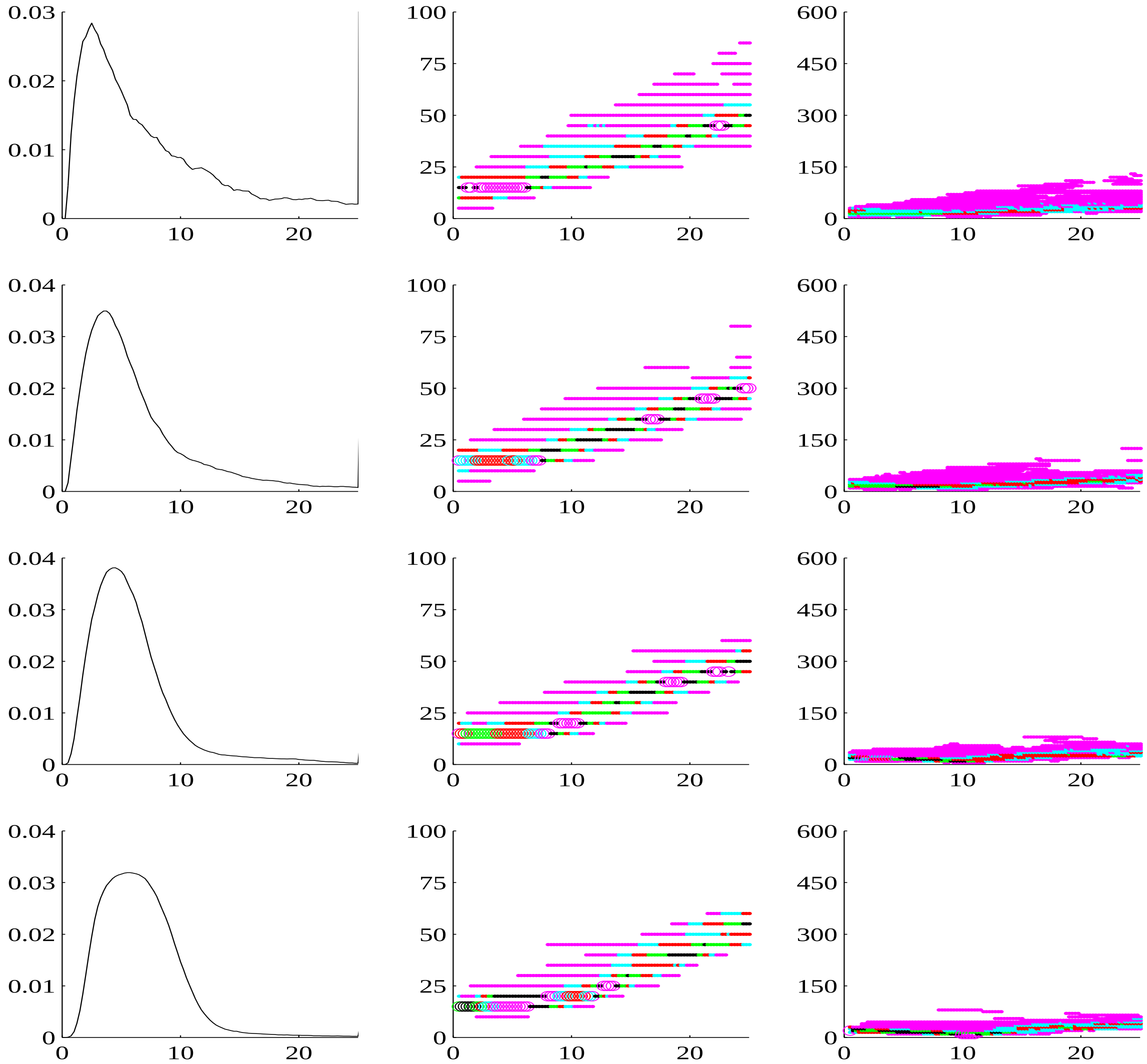


Figure A.2: Graphs show runs with 4 different loads and parameters ($O = 5$, $I = 5$, $t_a = 10\ ms$, $t_{rs} = .1\ ms$, $t_{ss} = .1\ ms$).

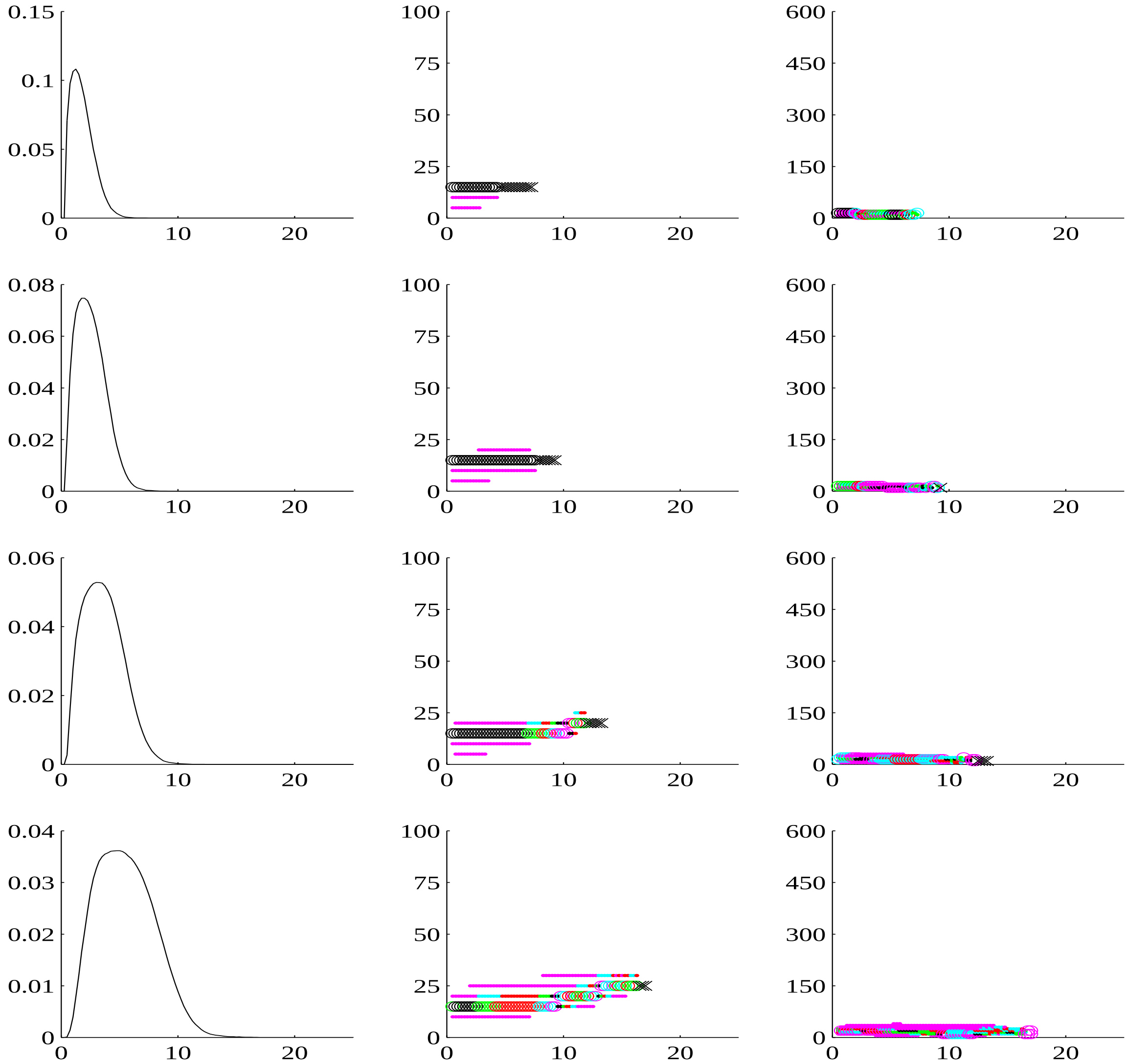


Figure A.3: Graphs show runs with 4 different loads and parameters (O = 5, I = 5, t_a = 10 ms, t_{rs} = .1 ms, t_{ss} = .5 s).

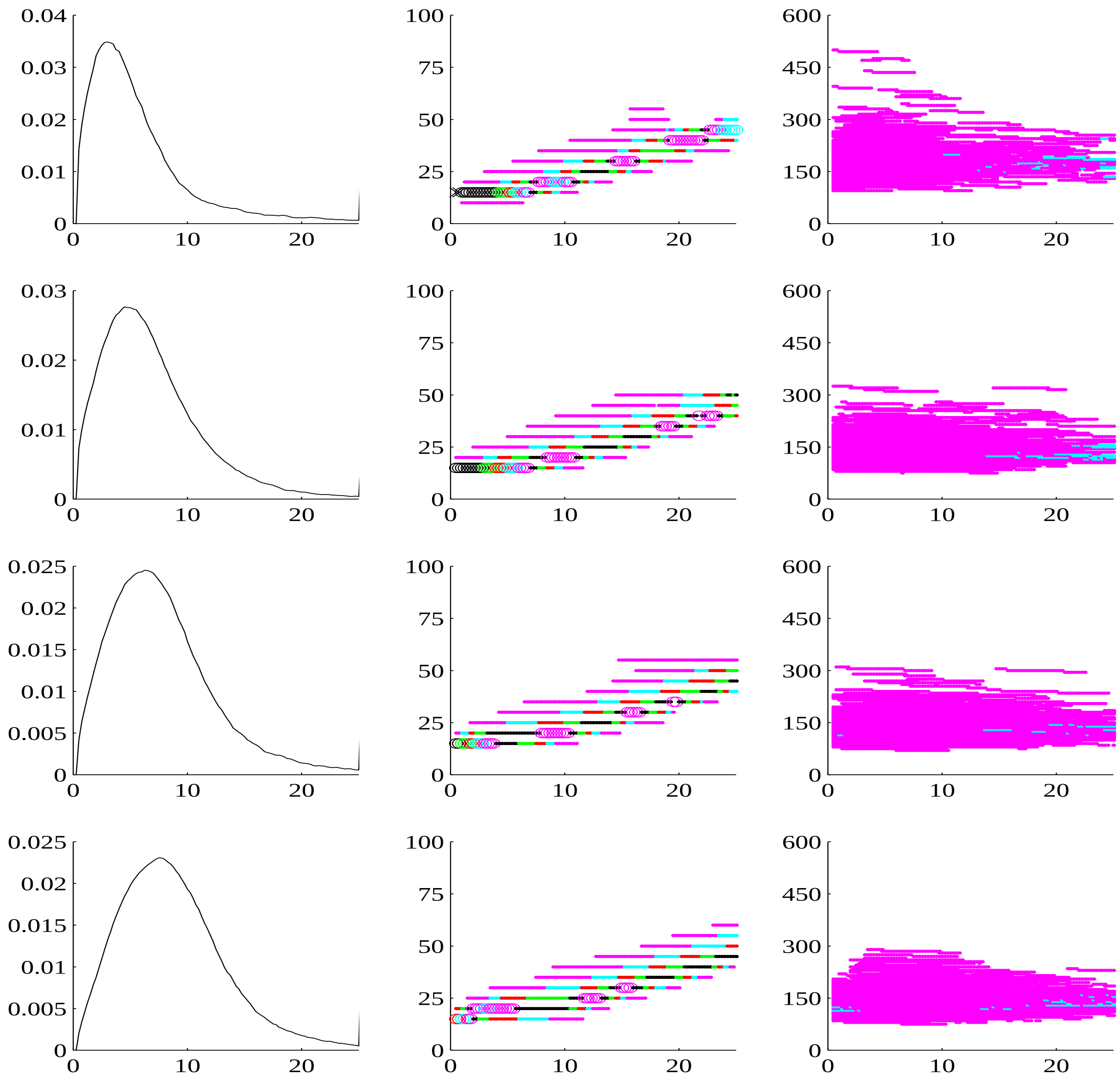


Figure A.4: Graphs show runs with 4 different loads and parameters ($O = 5$, $I = 5$, $t_a = 10\ ms$, $t_{rs} = .5\ s$, $t_{ss} = .1\ ms$).

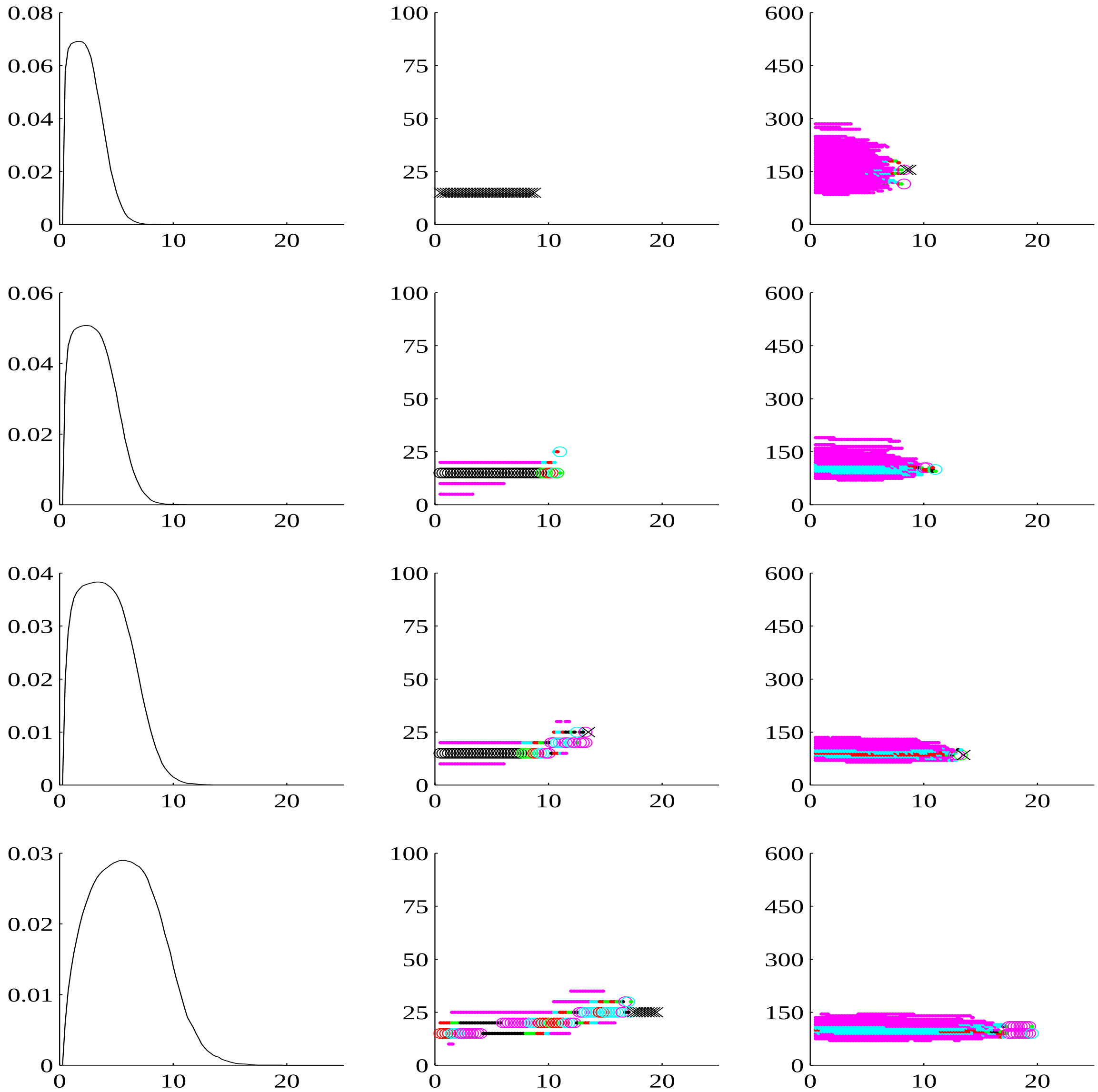


Figure A.5: Graphs show runs with 4 different loads and parameters (O = 5, I = 5, t_a = 10 ms, t_{rs} = .5 s, t_{ss} = .5 s).

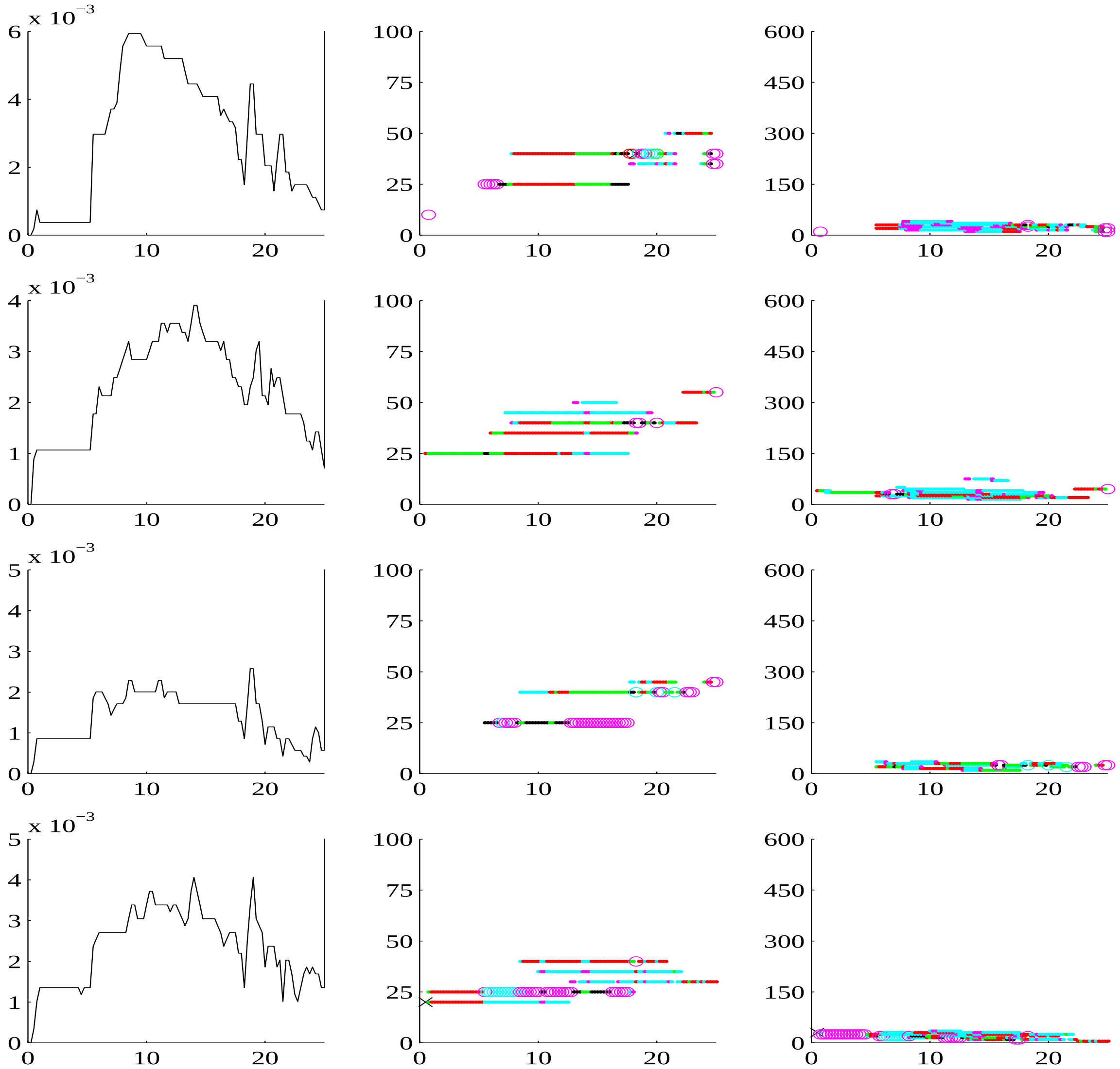


Figure A.6: Graphs show runs with 4 different loads and parameters ($O = 5$, $I = 5$, $t_a = 1\ s$, $t_{rs} = .1\ ms$, $t_{ss} = .1\ ms$).

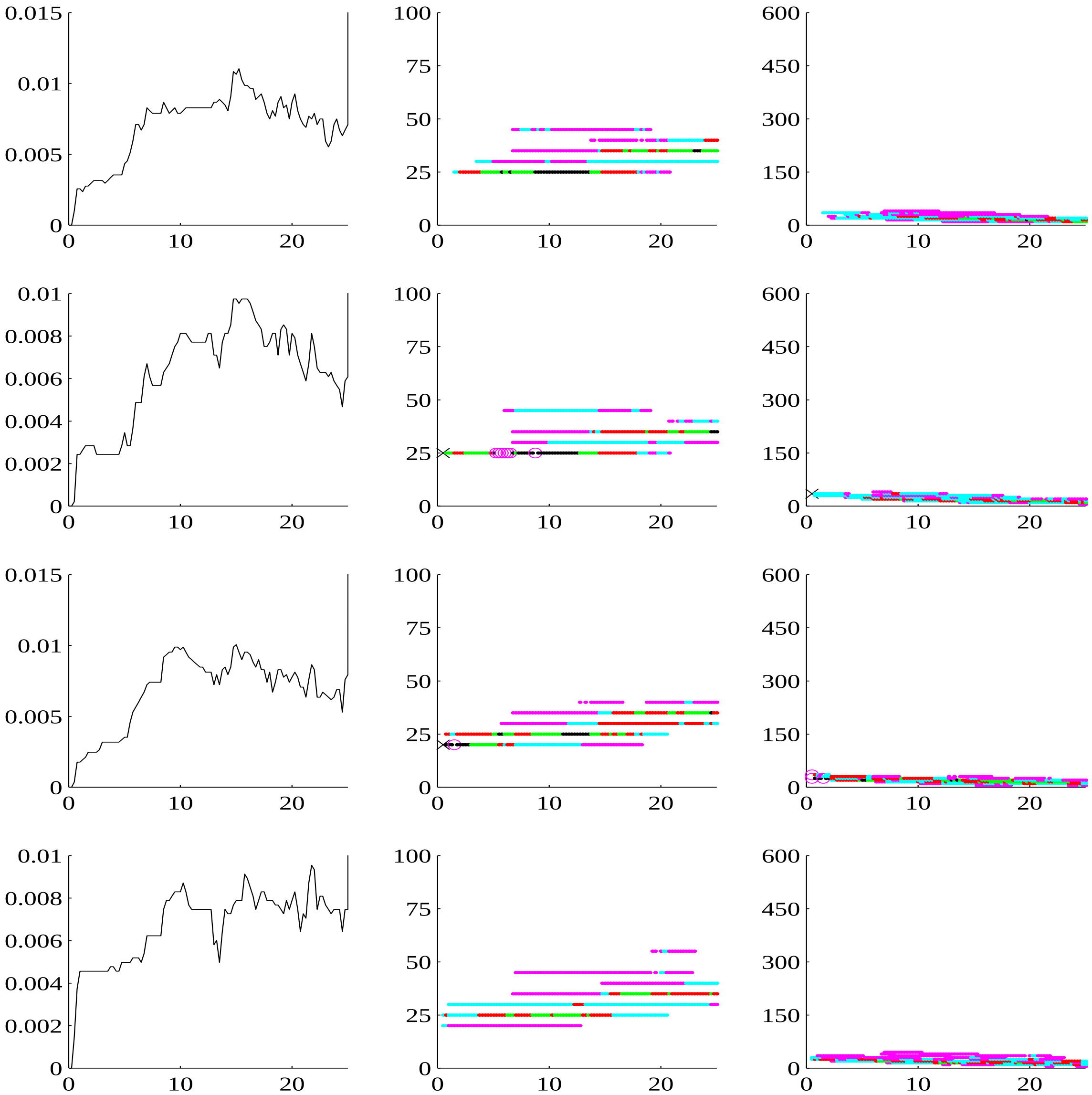


Figure A.7: Graphs show runs with 4 different loads and parameters ($O = 5$, $I = 5$, $t_a = 1\ s$, $t_{rs} = .1\ ms$, $t_{ss} = .5\ s$).

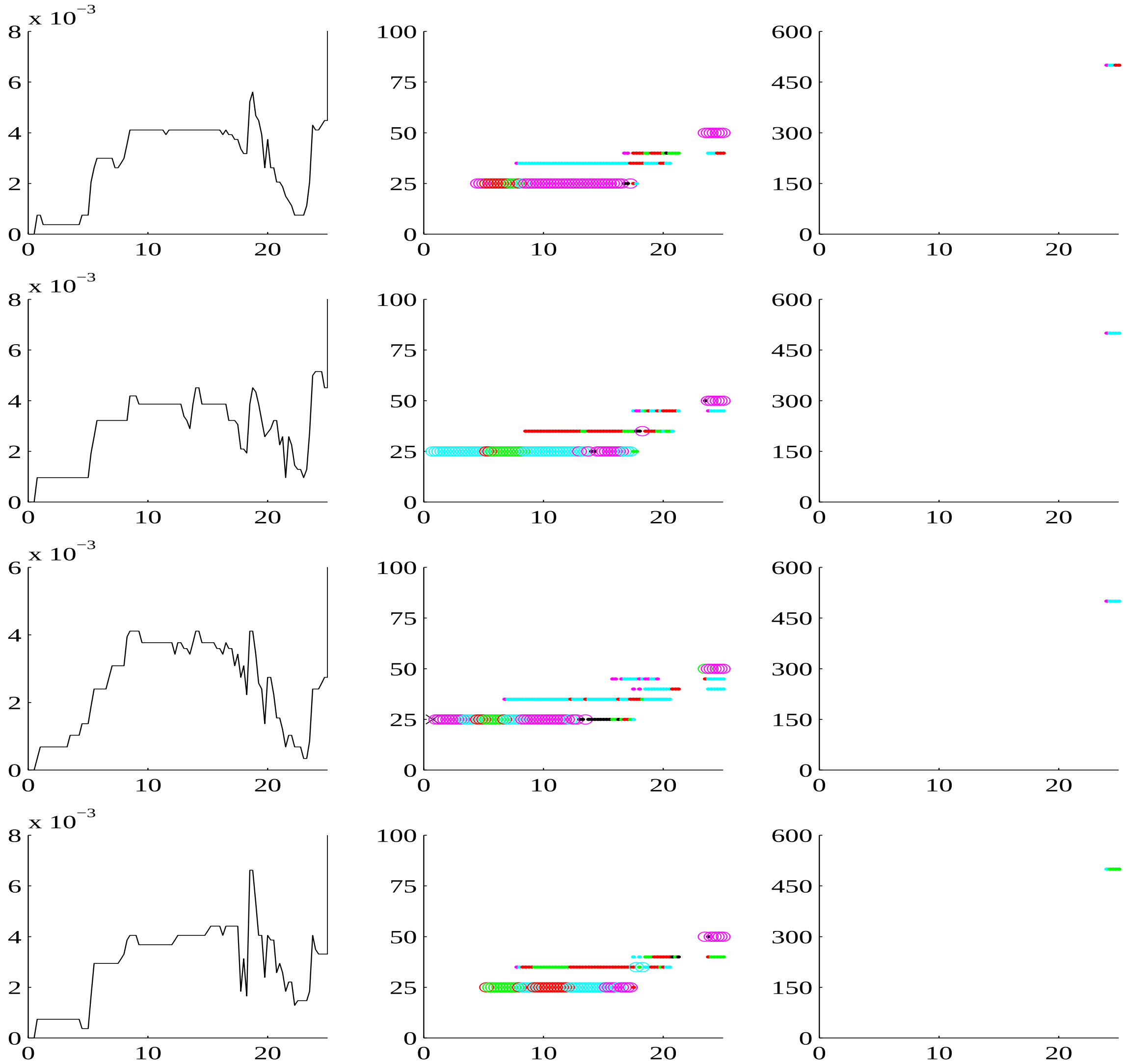


Figure A.8: Graphs show runs with 4 different loads and parameters ($O = 5$, $I = 5$, $t_a = 1\ s$, $t_{rs} = .5\ s$, $t_{ss} = .1\ ms$).

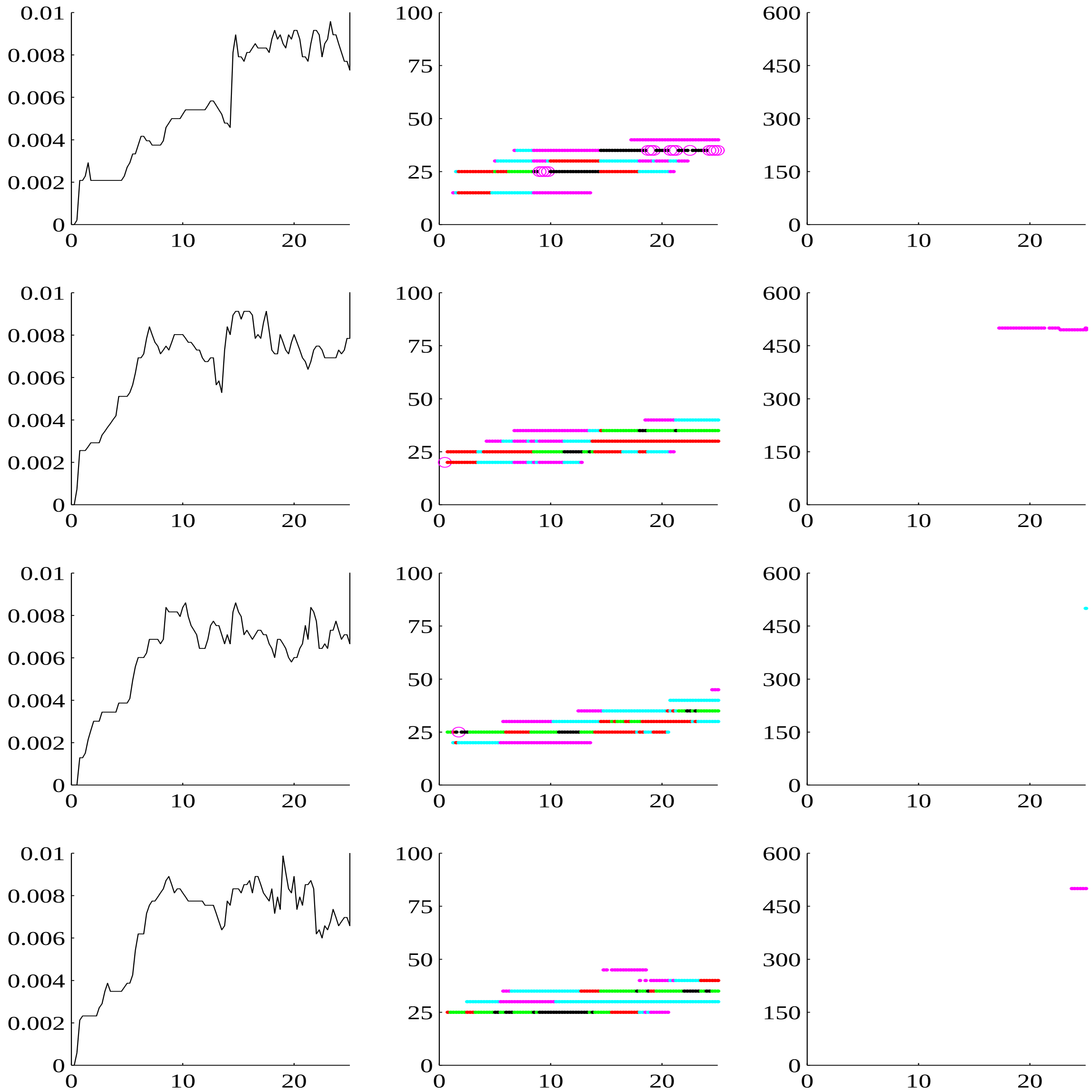


Figure A.9: Graphs show runs with 4 different loads and parameters ($O = 5$, $I = 5$, $t_a = 1\ s$, $t_{rs} = .5\ s$, $t_{ss} = .5\ s$).

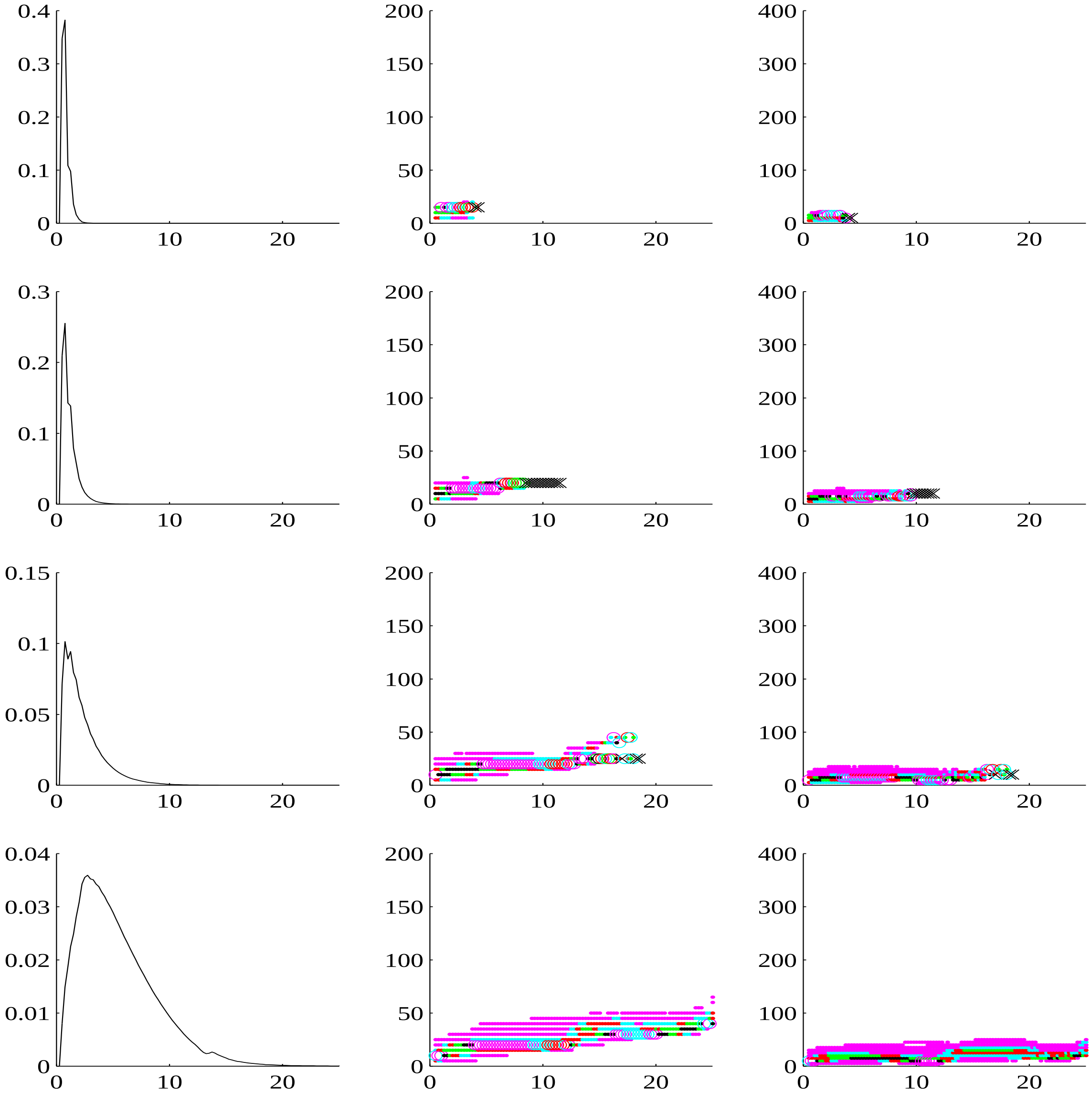


Figure A.10: Graphs show runs with 4 different loads and parameters (O = 5, I = 1, t_a = 10 ms, t_{rs} = .1 ms, t_{ss} = .1 ms).

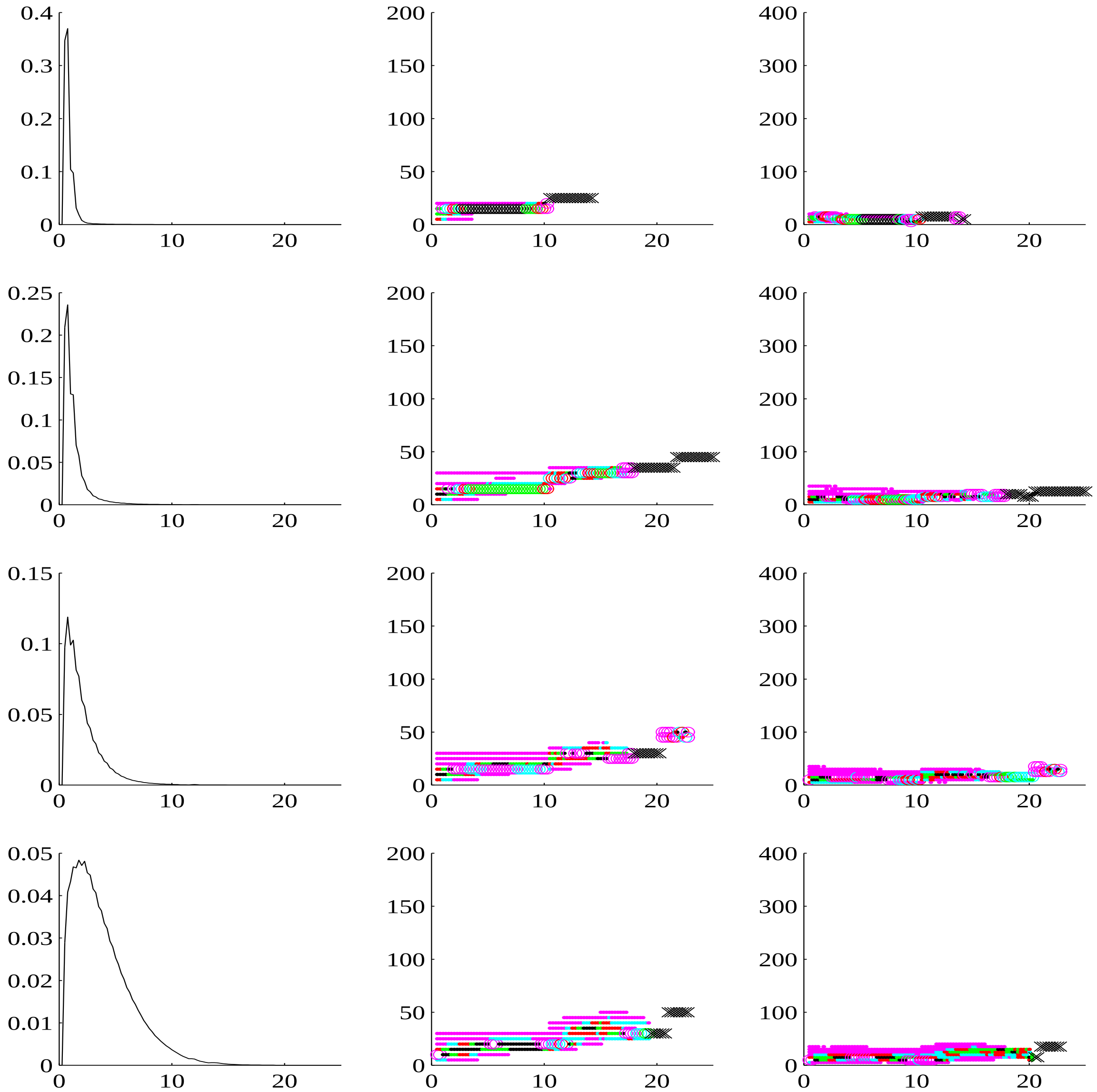


Figure A.11: Graphs show runs with 4 different loads and parameters (O = 5, I = 1, t_a = 10 ms, t_{rs} = .1 ms, t_{ss} = .5 s).

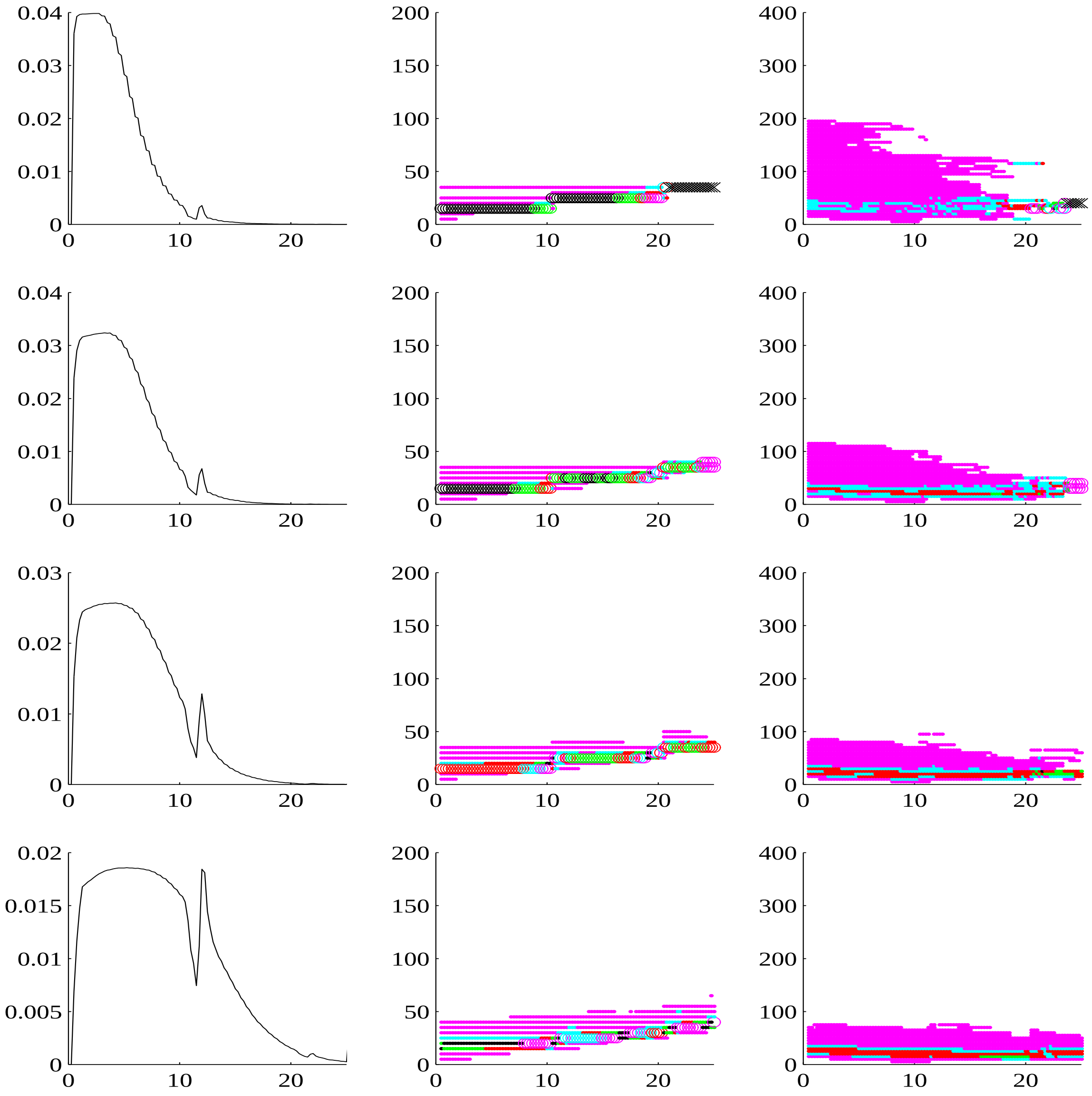


Figure A.12: Graphs show runs with 4 different loads and parameters (O = 5, I = 1, t_a = 10 ms, t_{rs} = .5 s, t_{ss} = .1 ms)

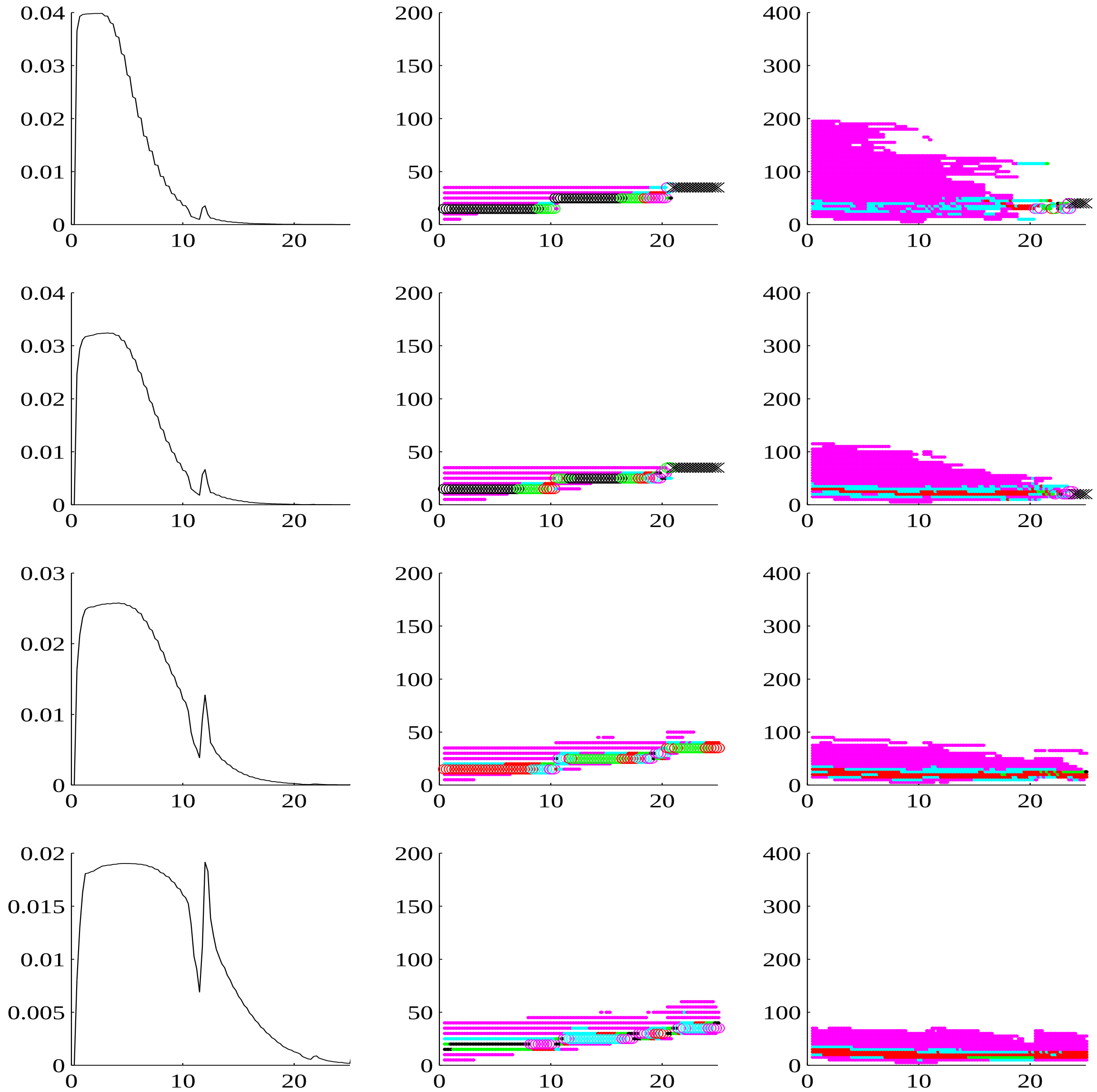


Figure A.13: Graphs are 4 different loads and parameters ($O = 5$, $I = 1$, $t_a = 10$ ms, $t_{rs} = .5\ s$, $t_{ss} = .5\ s$).

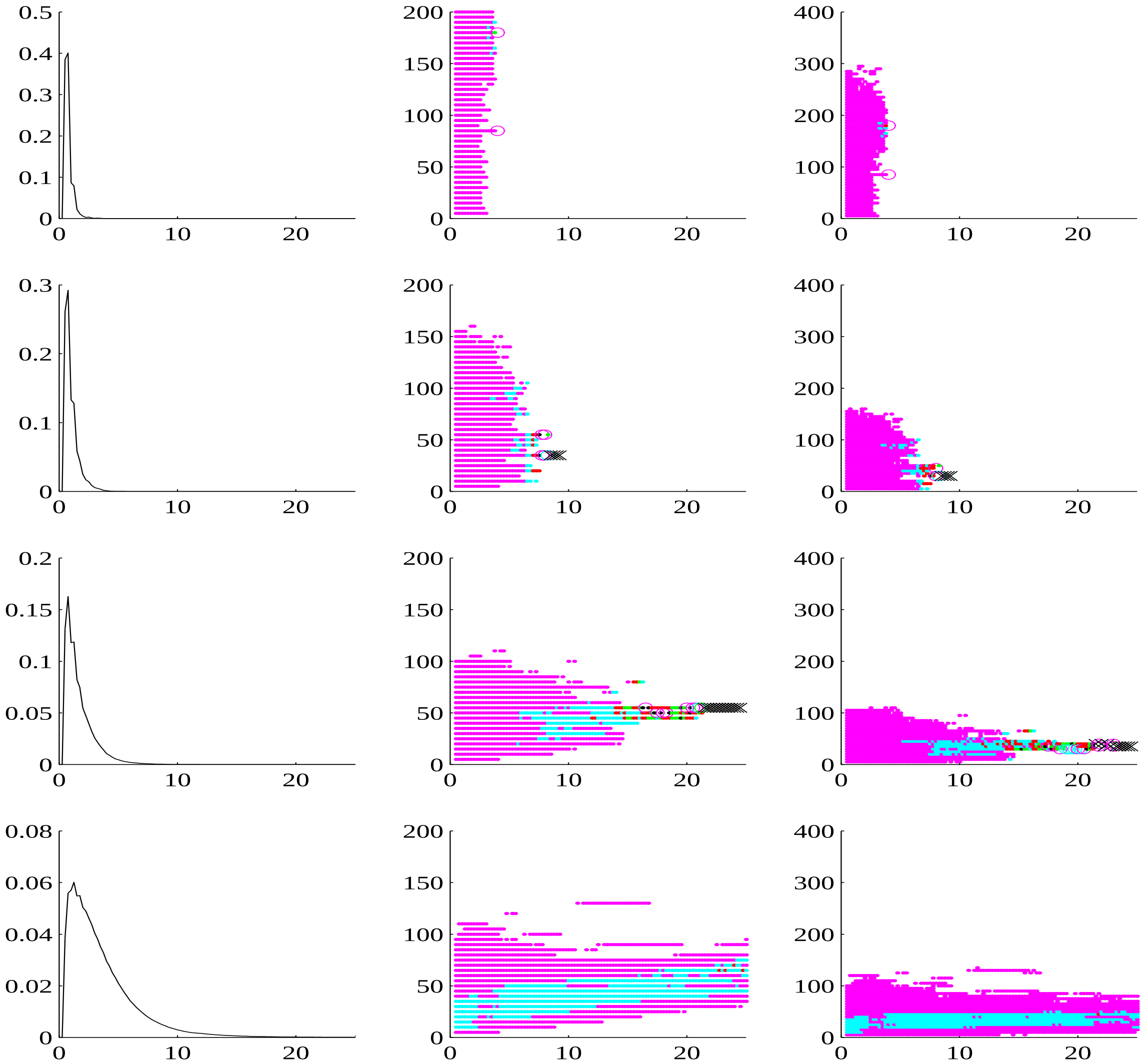


Figure A.14: Graphs show runs with 4 different loads and parameters (O = 5, I = 1, $t_a = 1$ s, $t_{rs} = .1$ ms, $t_{ss} = .1$ ms).

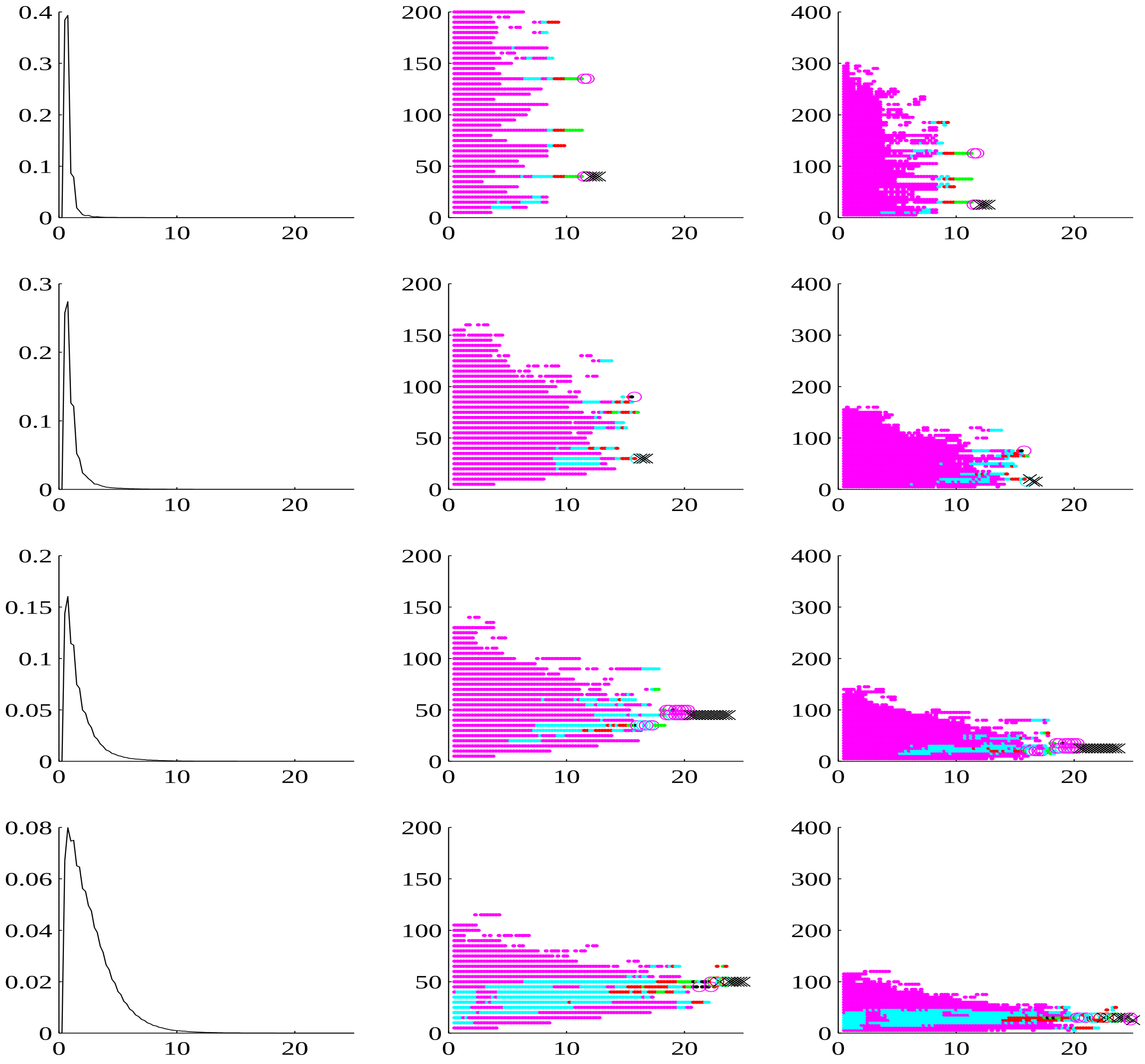


Figure A.15: Graphs show runs with 4 different loads and parameters (O = 5, I = 1, t_a = 1 s, t_{rs} = .1 ms, t_{ss} = .5 s).

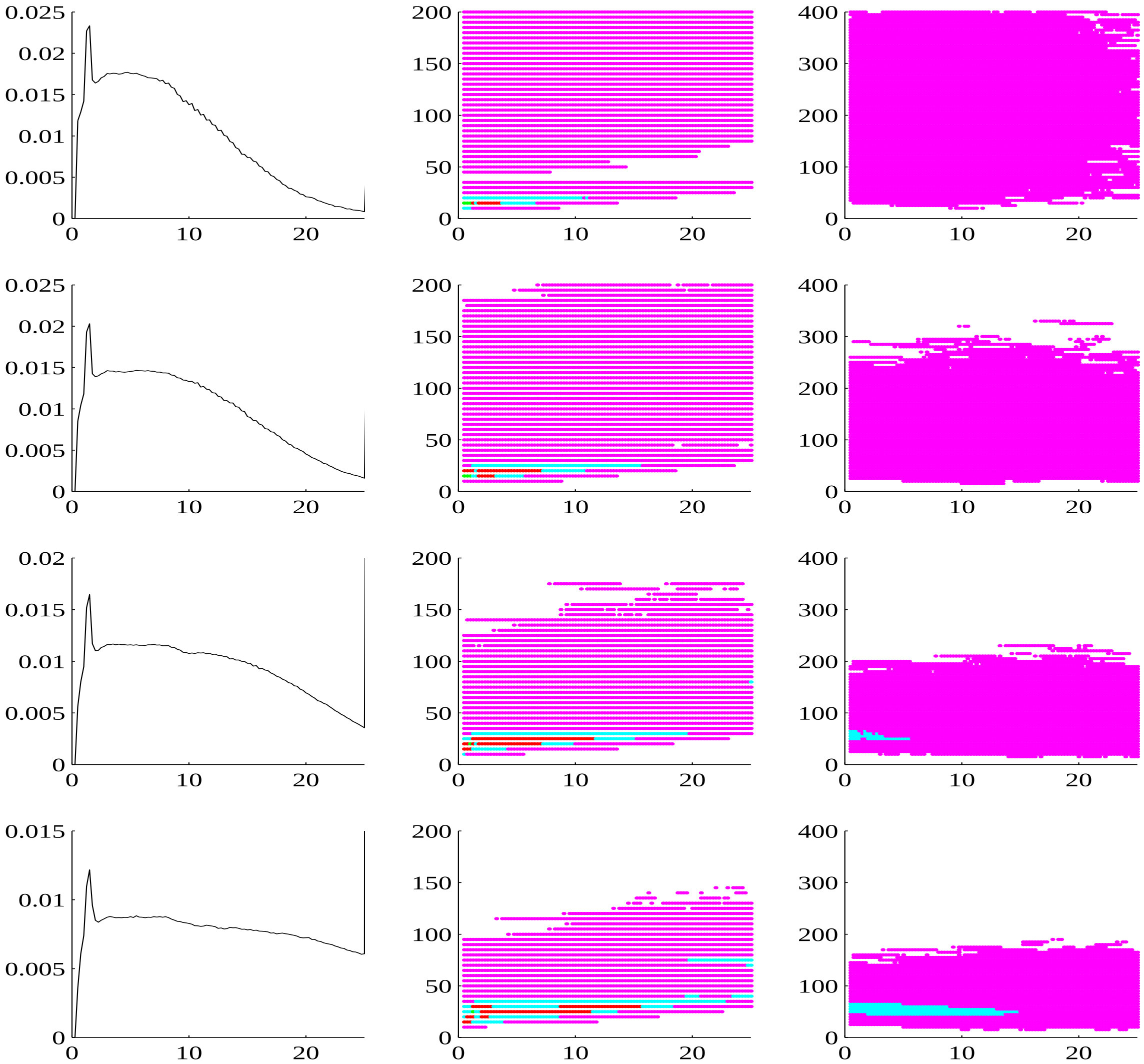


Figure A.16: Graphs show runs with 4 different loads and parameters (O = 5, I = 1, $t_a = 1$ s, $t_{rs} = .5$ s, $t_{ss} = .1$ ms).

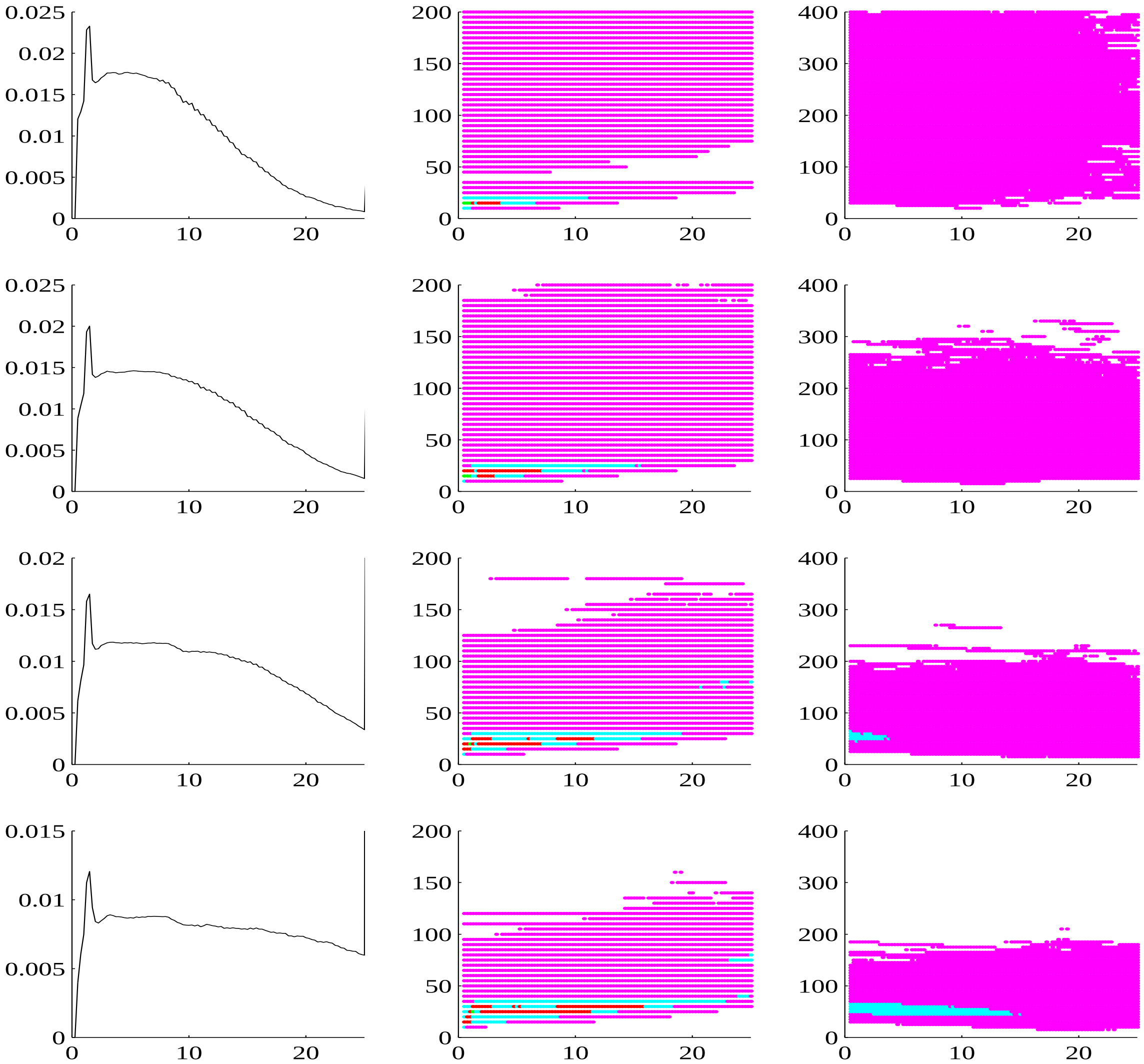


Figure A.17: Graphs show runs with 4 different loads and parameters (O = 5, I = 1, t_a = 1 s, t_{rs} = .5 s, t_{ss} = .5 s).

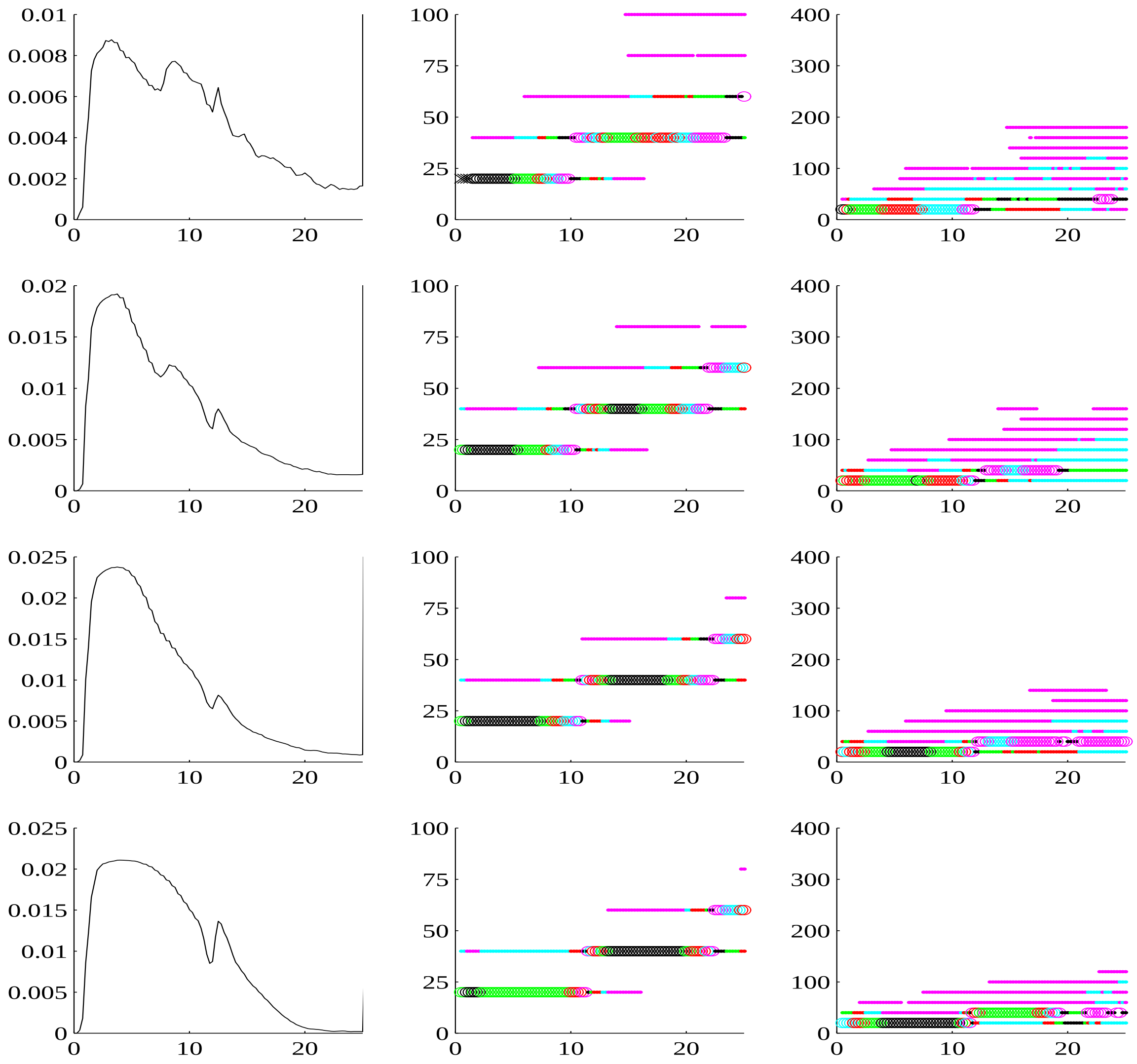


Figure A.18: Graphs show runs with 4 different loads and parameters (O = 1, I = 5, t_a = 10 ms, t_{rs} = .1 ms, t_{ss} = .1 ms).

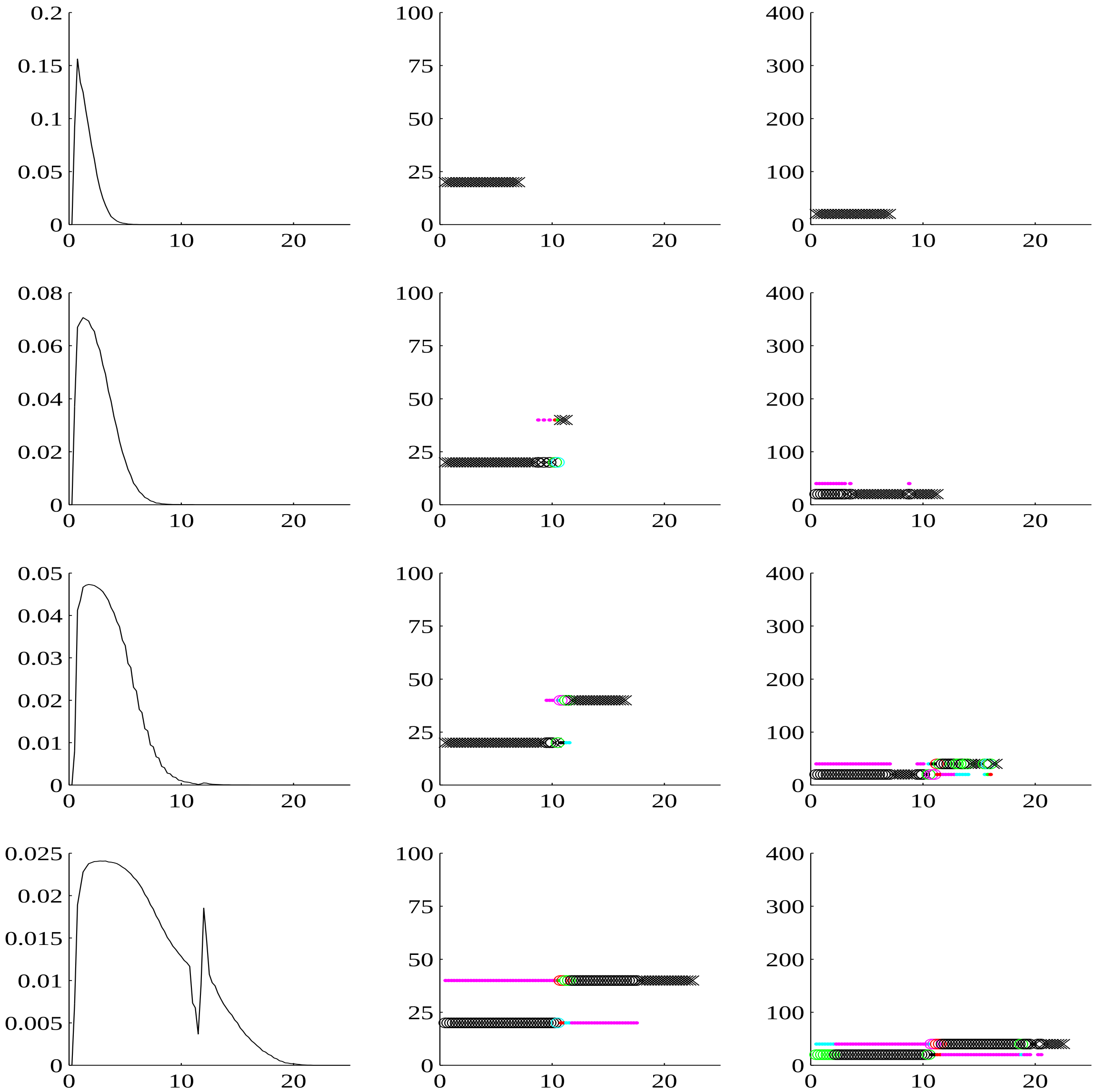


Figure A.19: Graphs show runs with 4 different loads and parameters (O = 1, I = 5, t_a = 10 ms, t_{rs} = .1 ms, t_{ss} = .5 s).

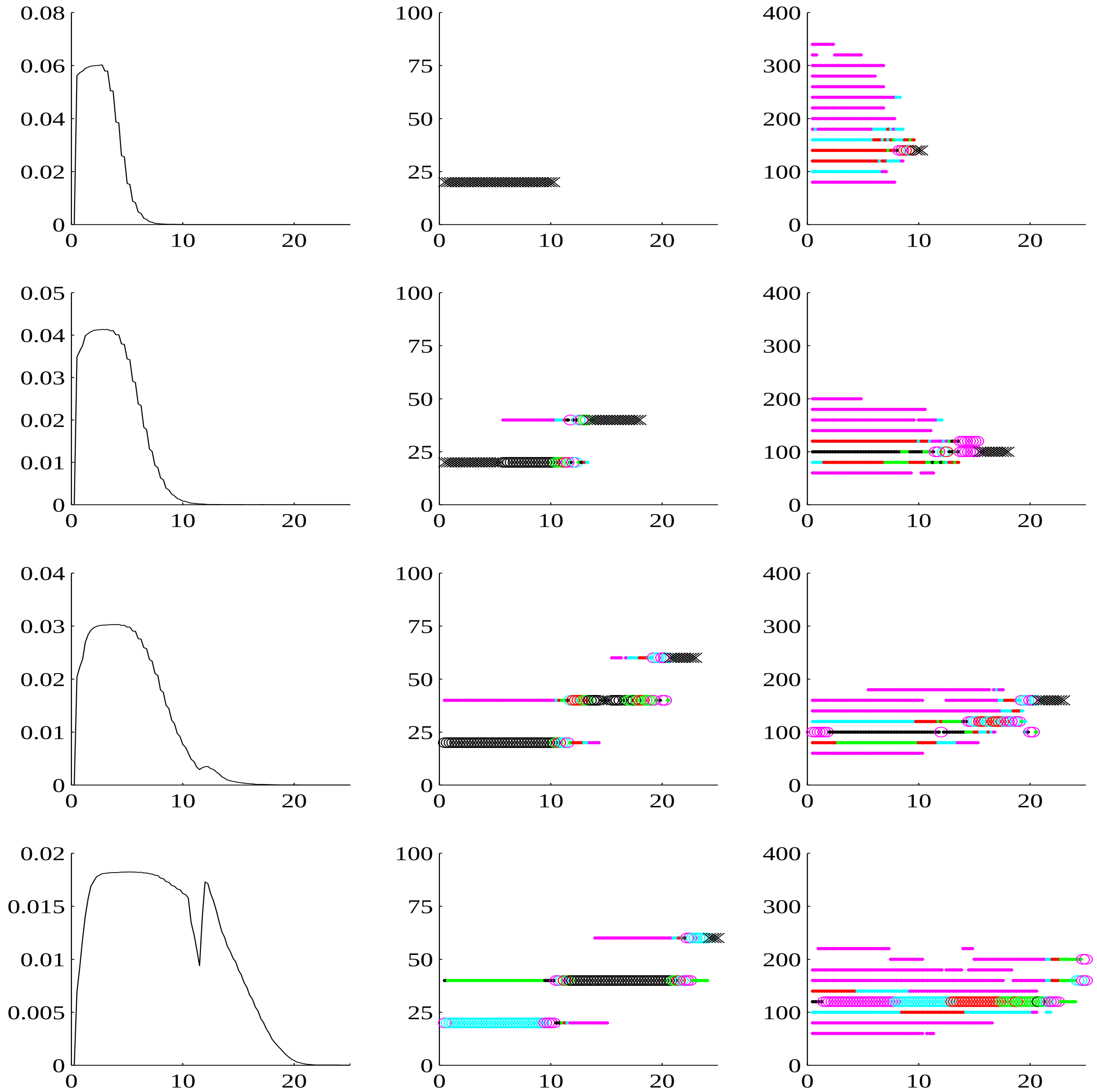


Figure A.20: Graphs show runs with 4 different loads and parameters (O = 1, I = 5, t_a = 10 ms, t_{rs} = .5 s, t_{ss} = .1 ms).

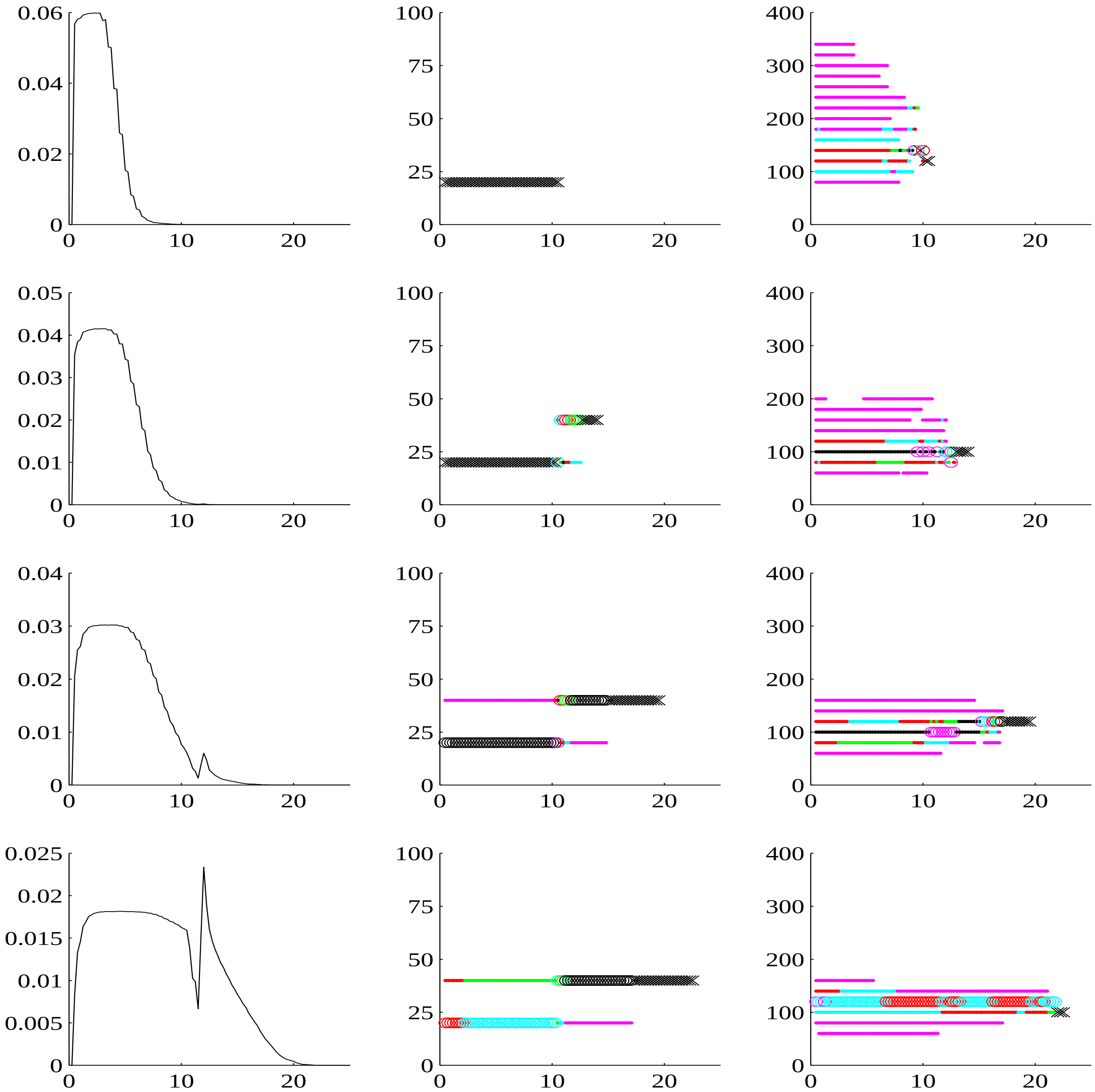


Figure A.21: Graphs show runs with 4 different loads and parameters (O = 1, I = 5, t_a = 10 ms, t_{rs} = .5 s, t_{ss} = .5 s).

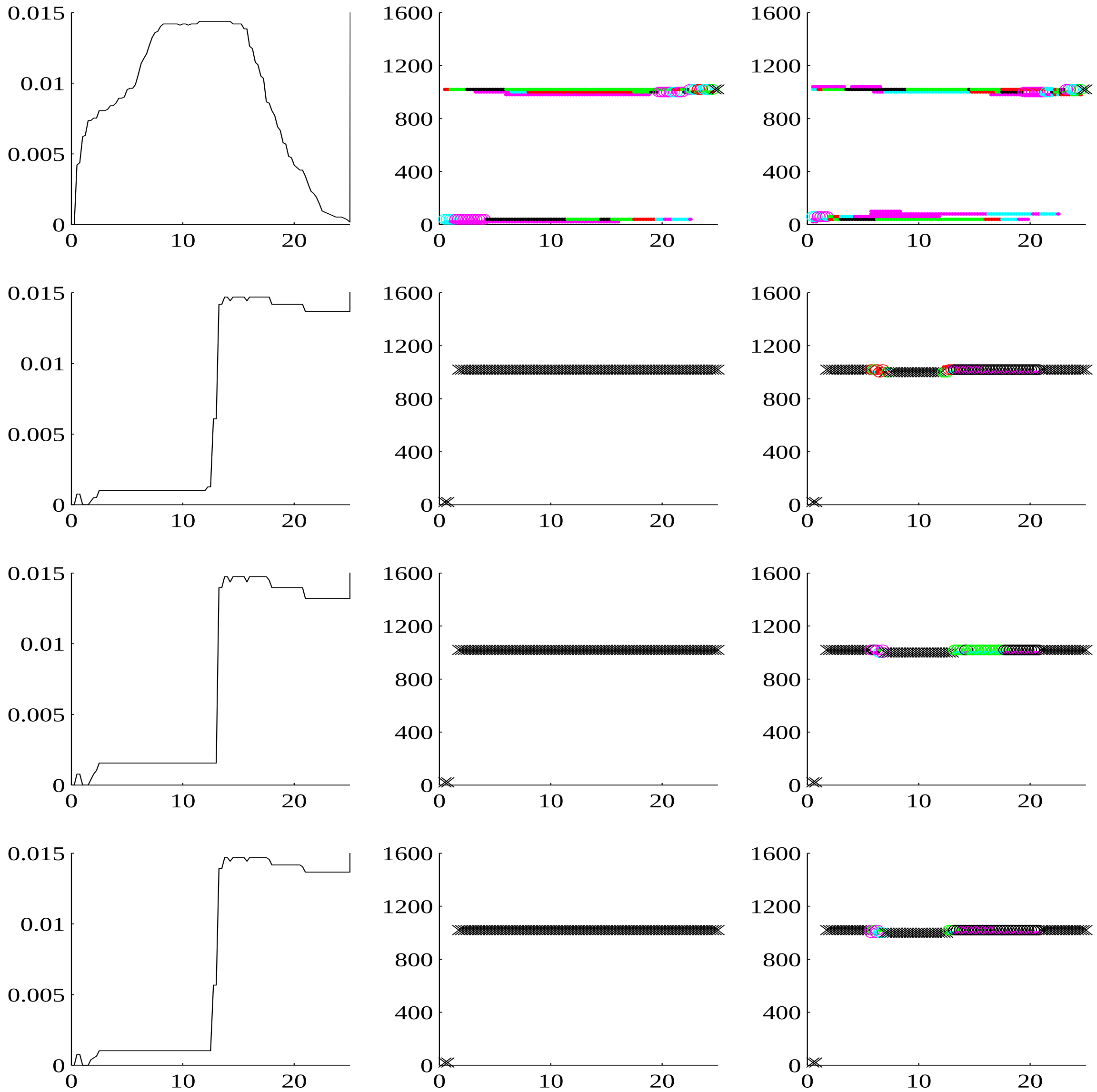


Figure A.22: Graphs show runs with 4 different loads and parameters (O = 1, I = 5, $t_a = 1\ s$, $t_{rs} = .1\ ms$, $t_{ss} = .1\ ms$).

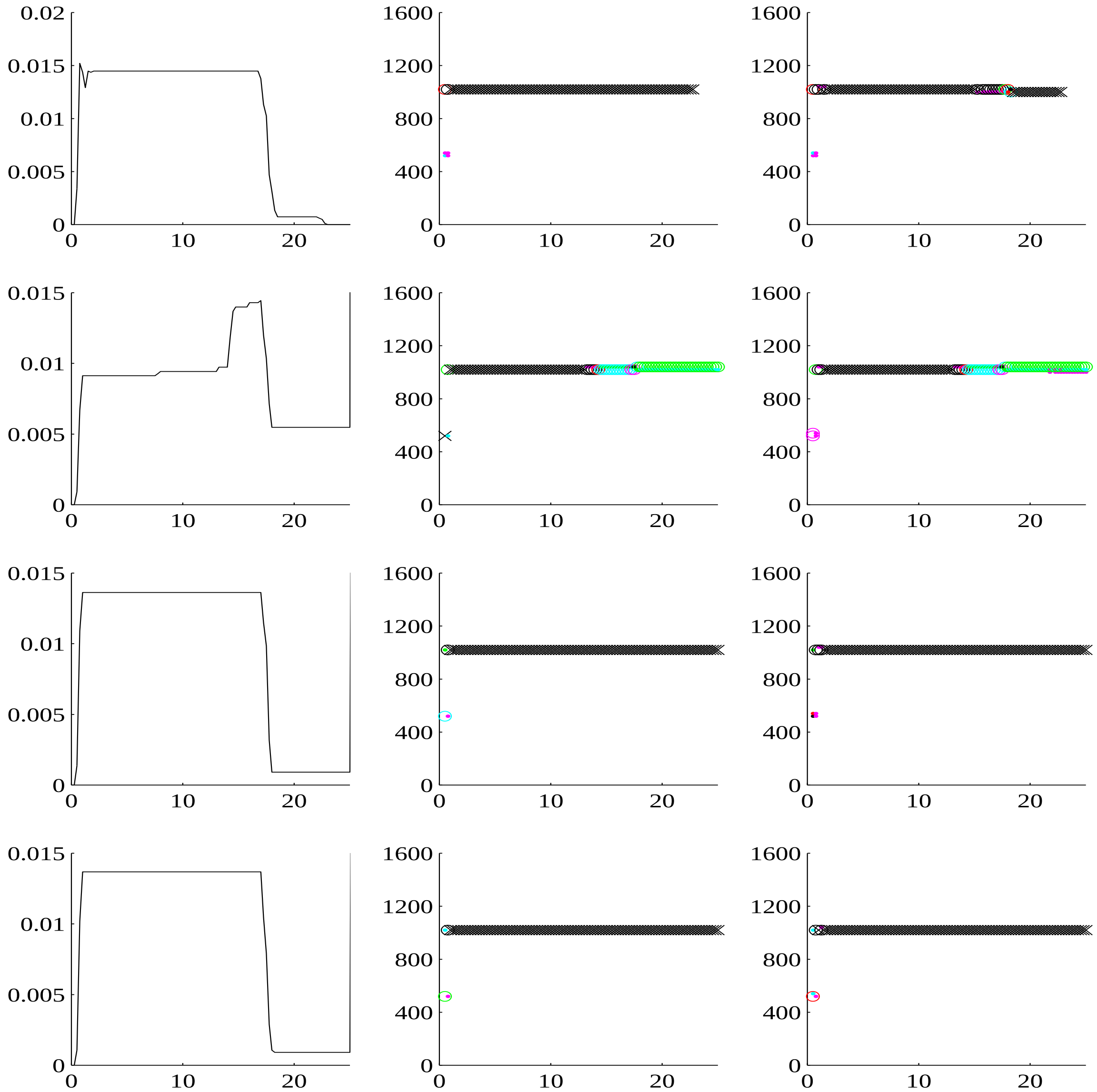


Figure A.23: Graphs show runs with 4 different loads and parameters (O = 1, I = 5, t_a = 1 s, t_{rs} = .1 ms, t_{ss} = .5 s).

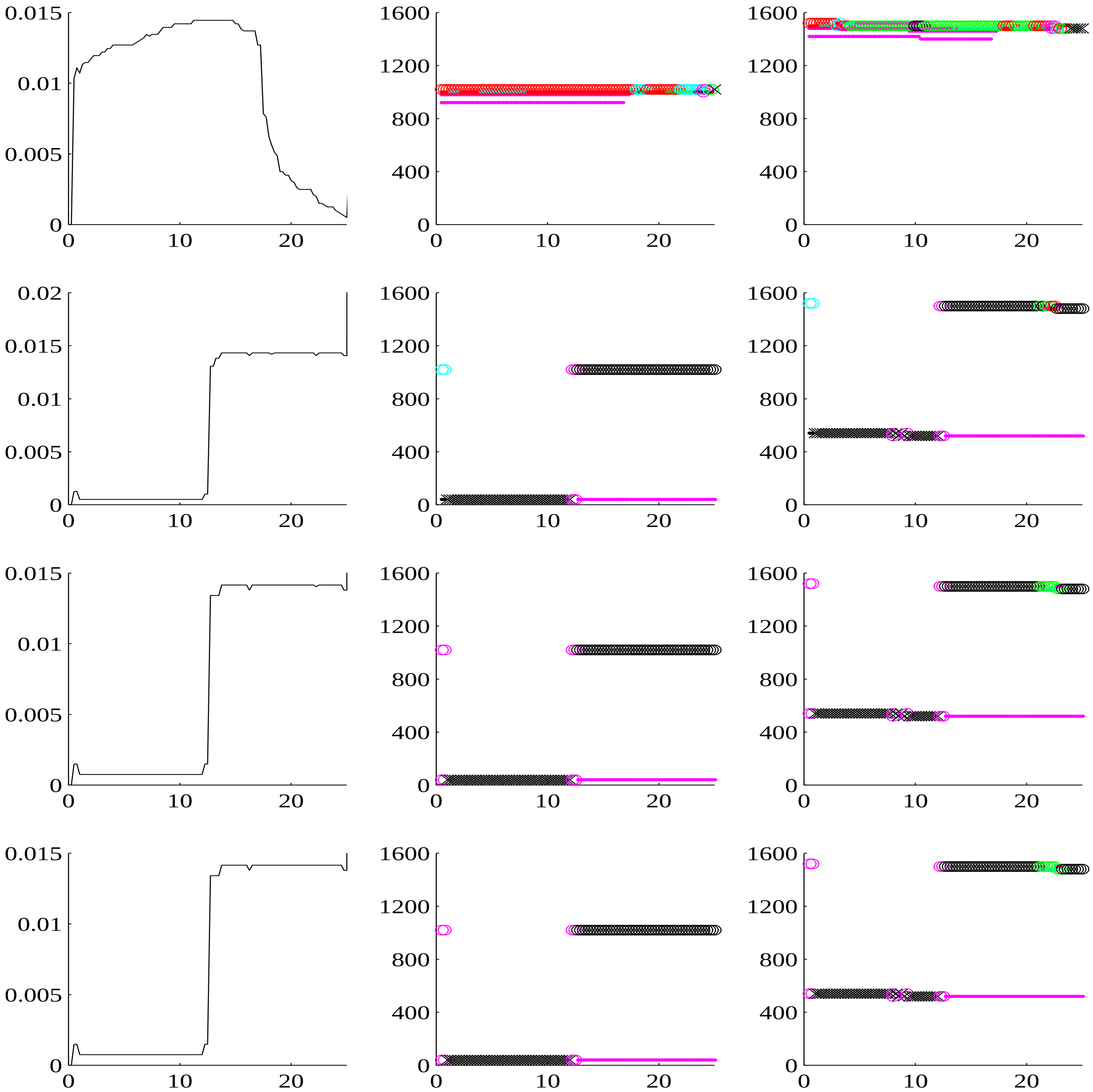


Figure A.24: Graphs show runs with 4 different loads and parameters (O = 1, I = 5, $t_a = 1\ s$, $t_{rs} = .5\ s$, $t_{ss} = .1\ ms$).

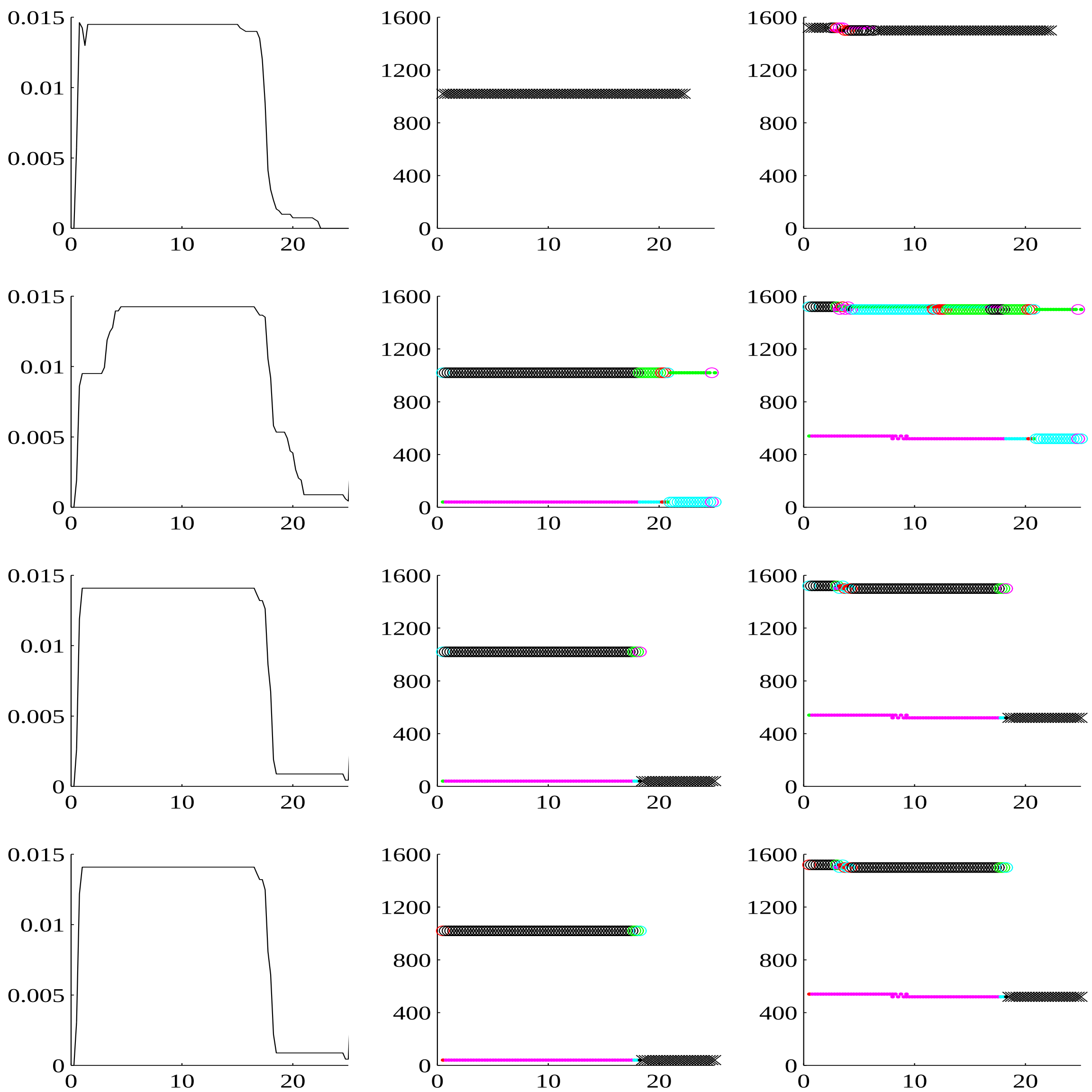


Figure A.25: Graphs show runs with 4 different loads and parameters (O = 1, I = 5, t_a = 1 s, t_{rs} = .5 s, t_{ss} = .5 s).

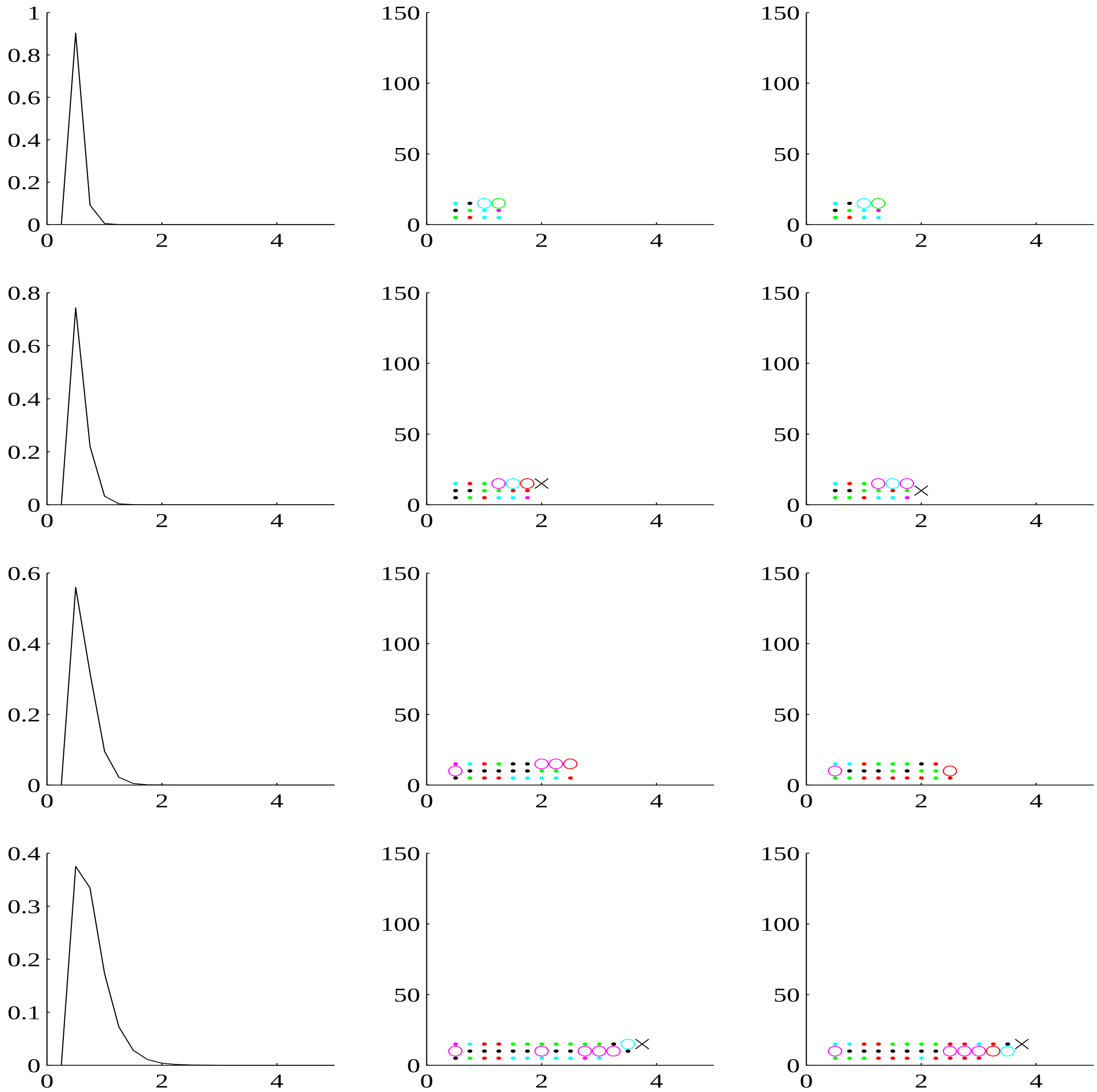


Figure A.26: Graphs show runs with 4 different loads and parameters (O = 1, I = 1, t_a = 10 ms, t_{rs} = .1 ms, t_{ss} = .1 ms).

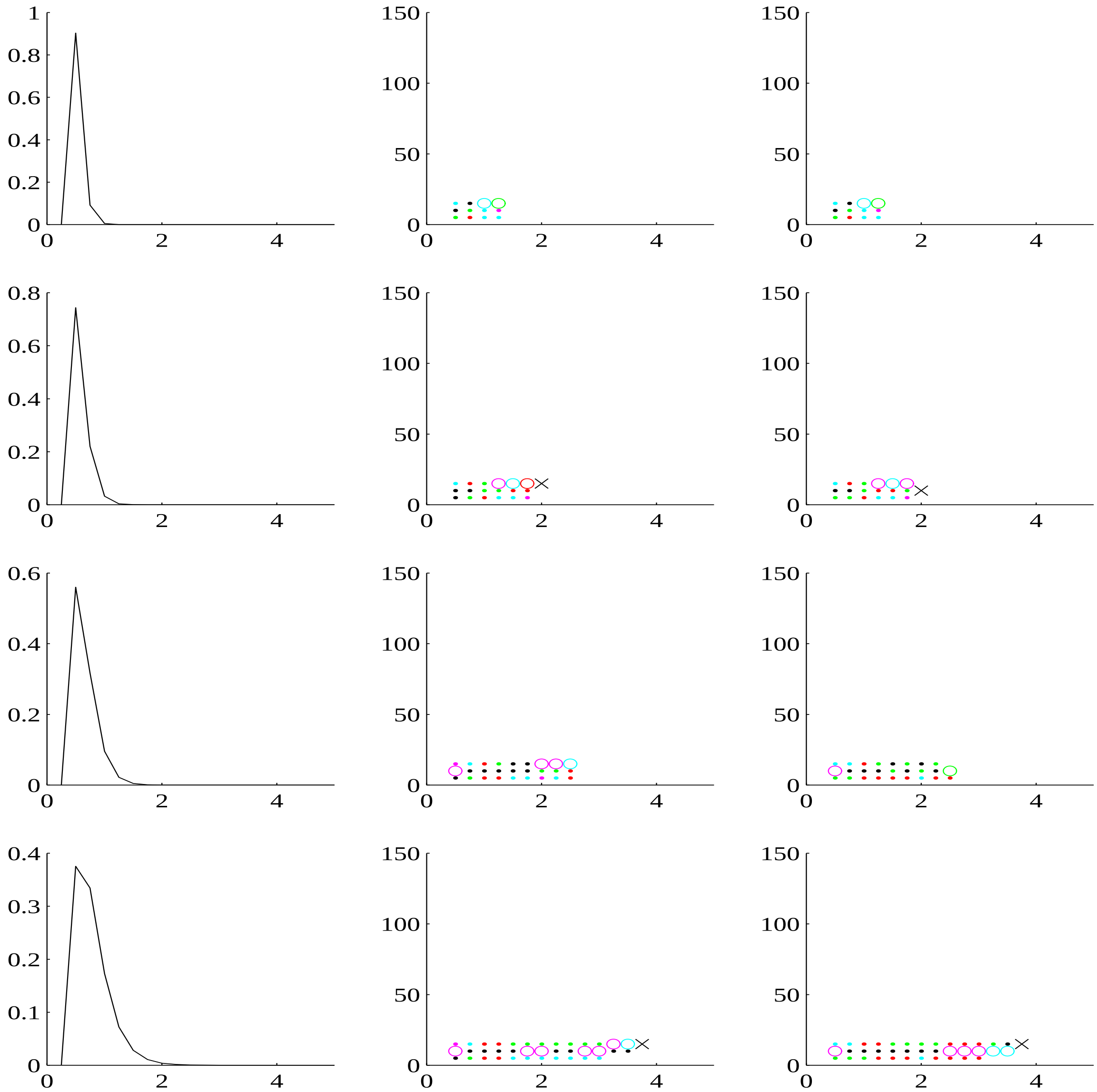


Figure A.27: Graphs show runs with 4 different loads and parameters (O = 1, I = 1, t_a = 10 ms, t_{rs} = .1 ms, t_{ss} = .5 s).

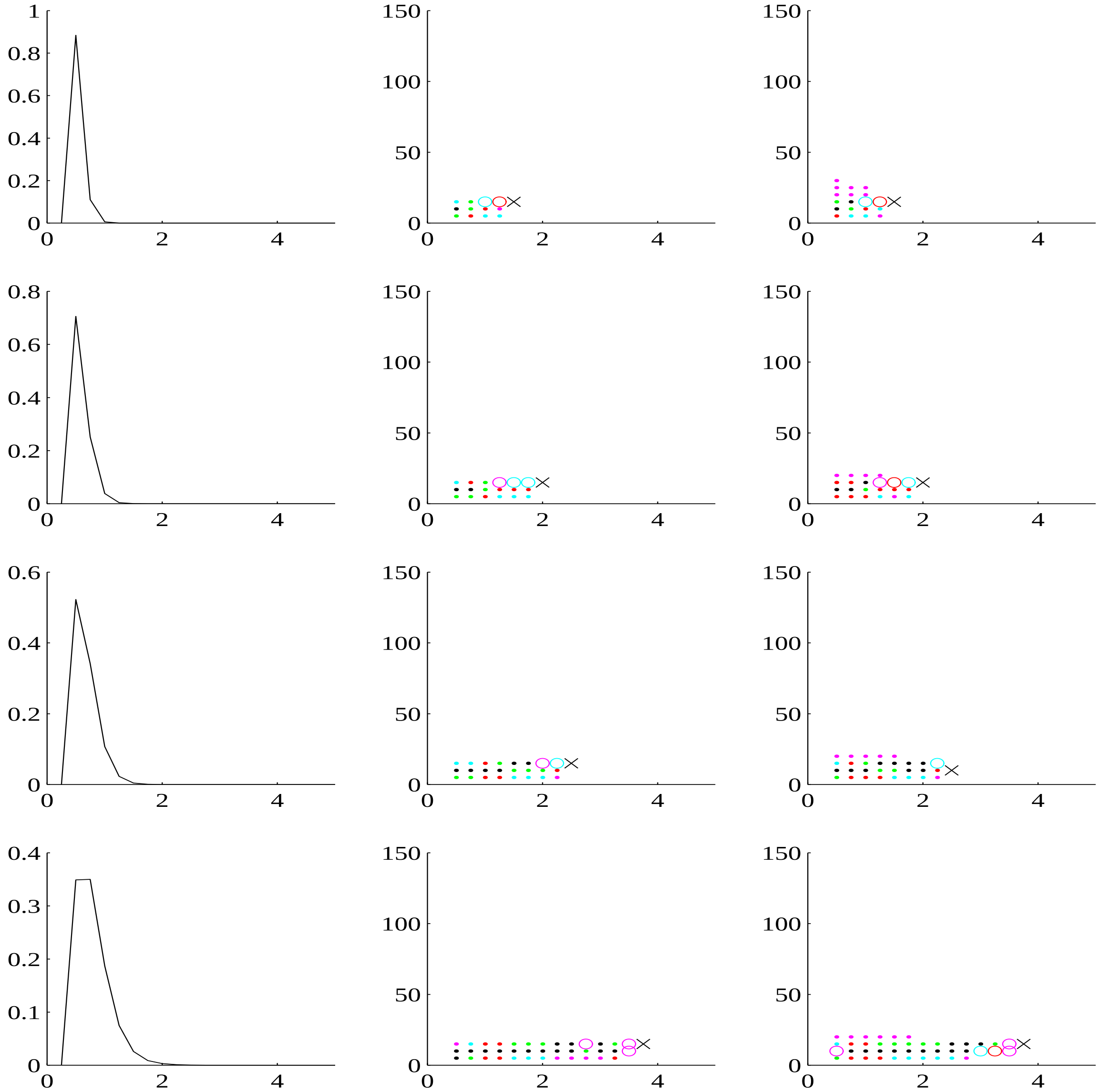


Figure A.28: Graphs show runs with 4 different loads and parameters (O = 1, I = 1, t_a = 10 ms, t_{rs} = .5 s, t_{ss} = .1 ms).

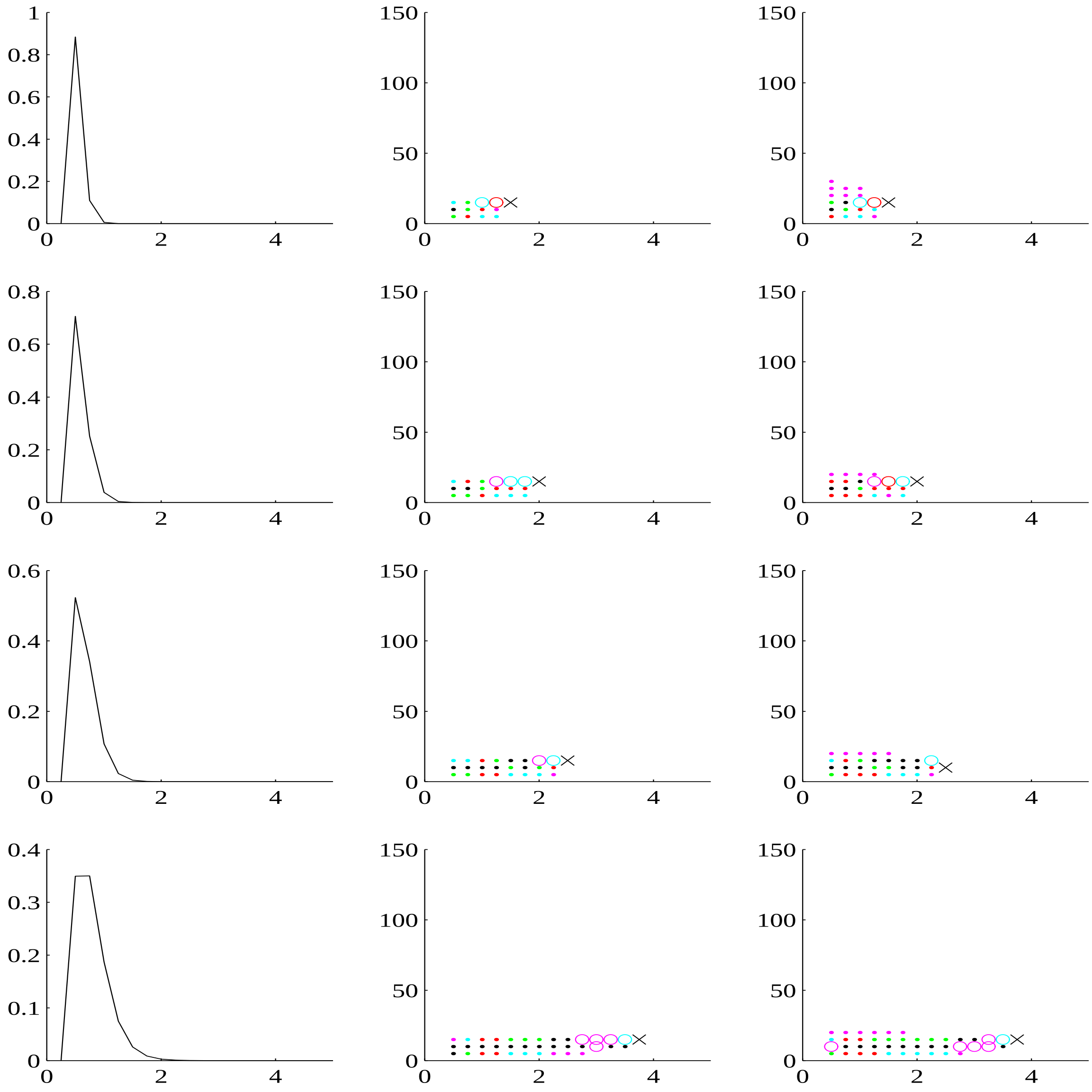


Figure A.29: Graphs show runs with 4 different loads and parameters (O = 1, I = 1, t_a = 10 ms, t_{rs} = .5 s, t_{ss} = .5 s).

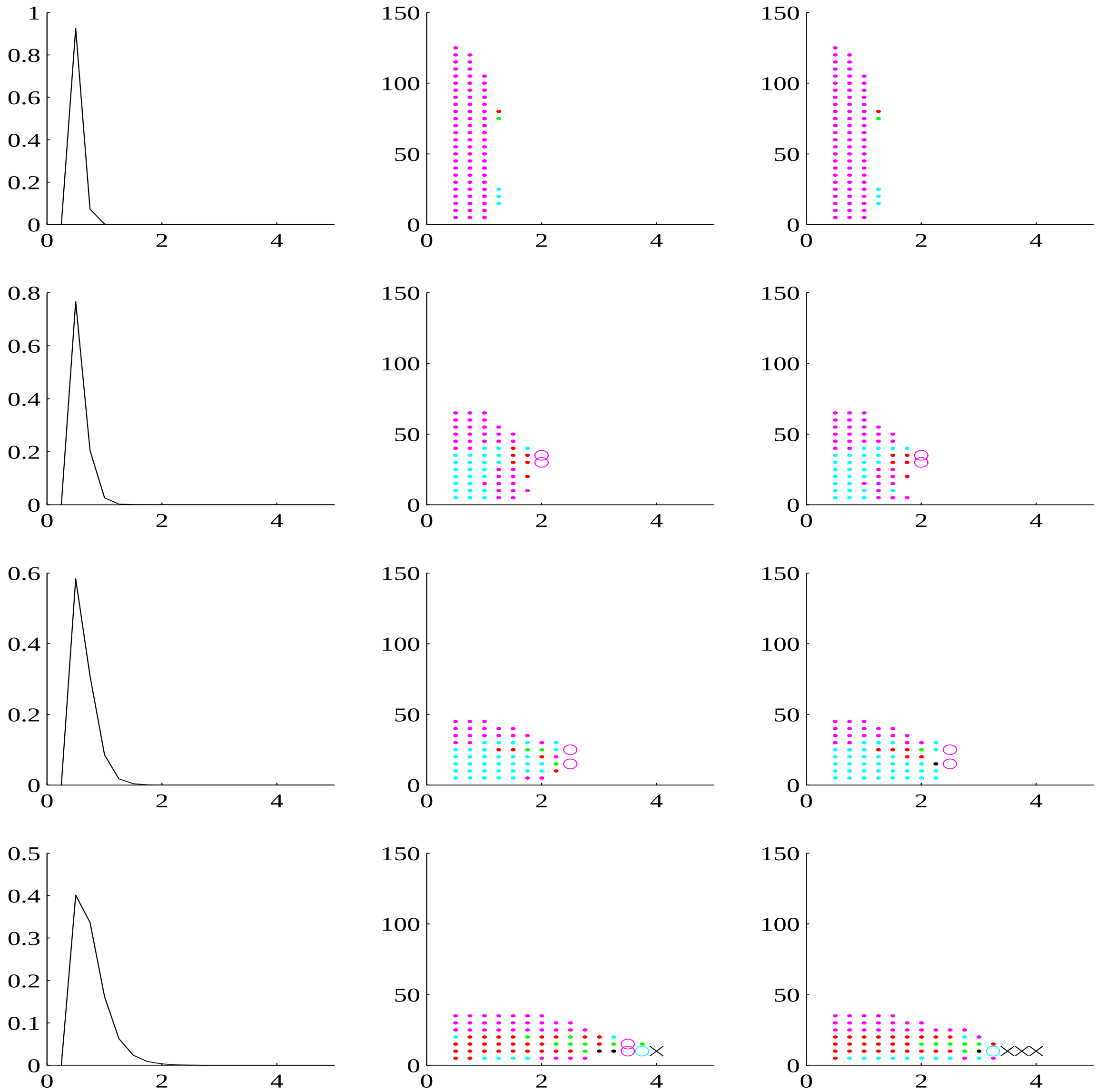


Figure A.30: Graphs show runs with 4 different loads and parameters (O = 1, I = 1, $t_a = 1$ s, $t_{rs} = .1$ ms, $t_{ss} = .1$ ms).

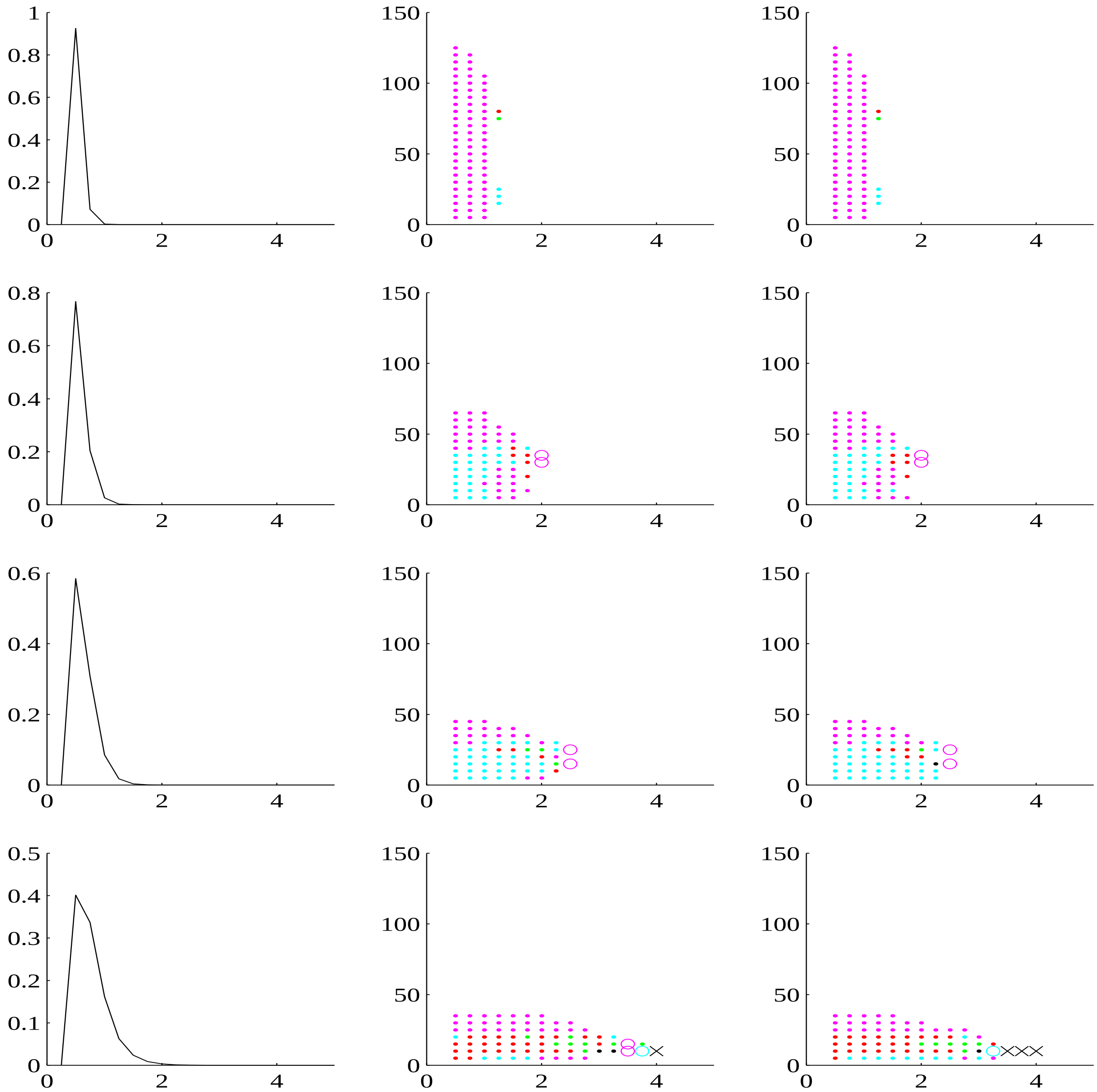


Figure A.31: Graphs show runs with 4 different loads and parameters (O = 1, I = 1, t_a = 1 s, t_{rs} = .1 ms, t_{ss} = .5 s).

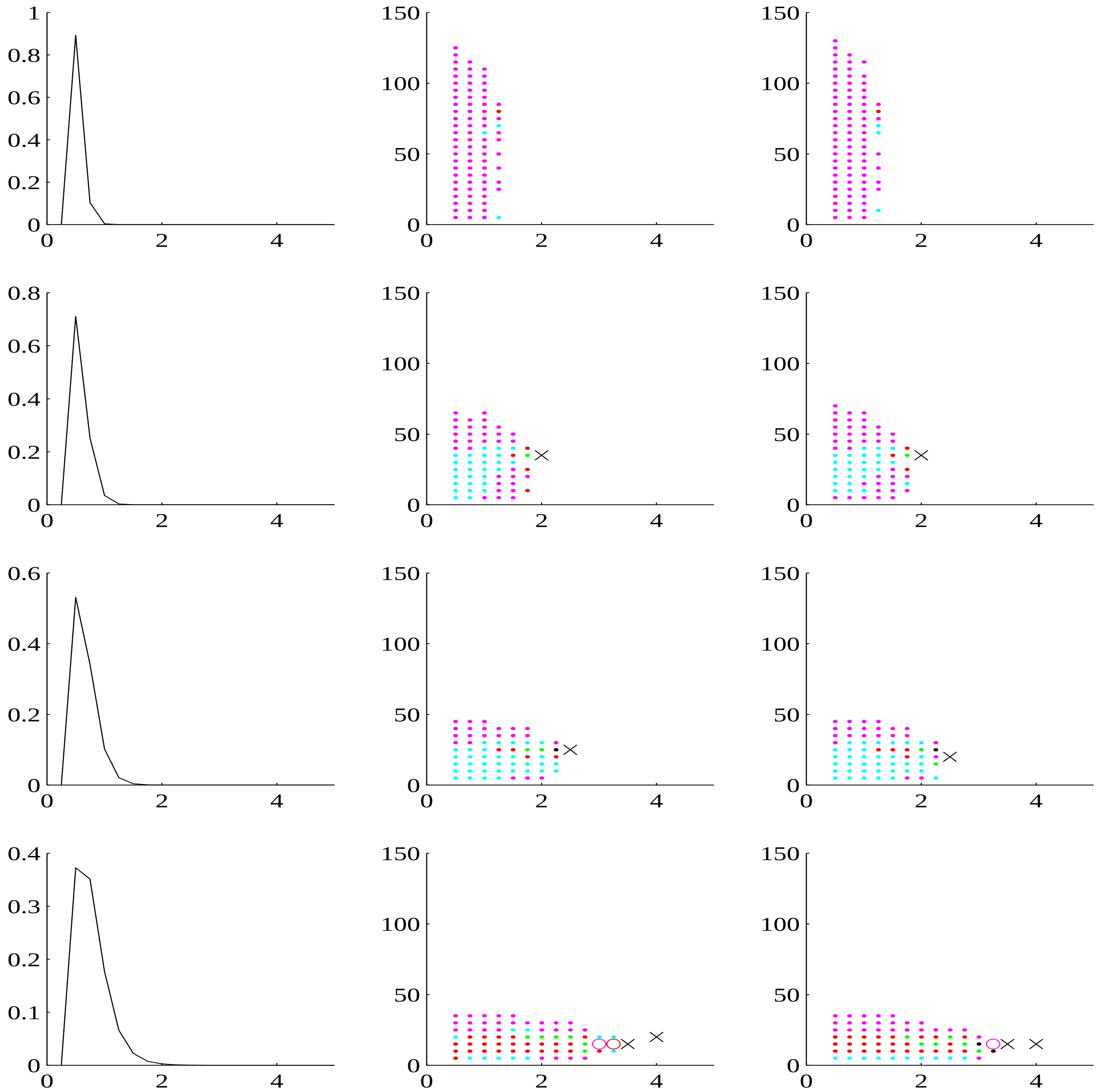


Figure A.32: Graphs show runs with 4 different loads and parameters (O = 1, I = 1, t_a = 1 s, t_{rs} = .5 s, t_{ss} = .1 ms).

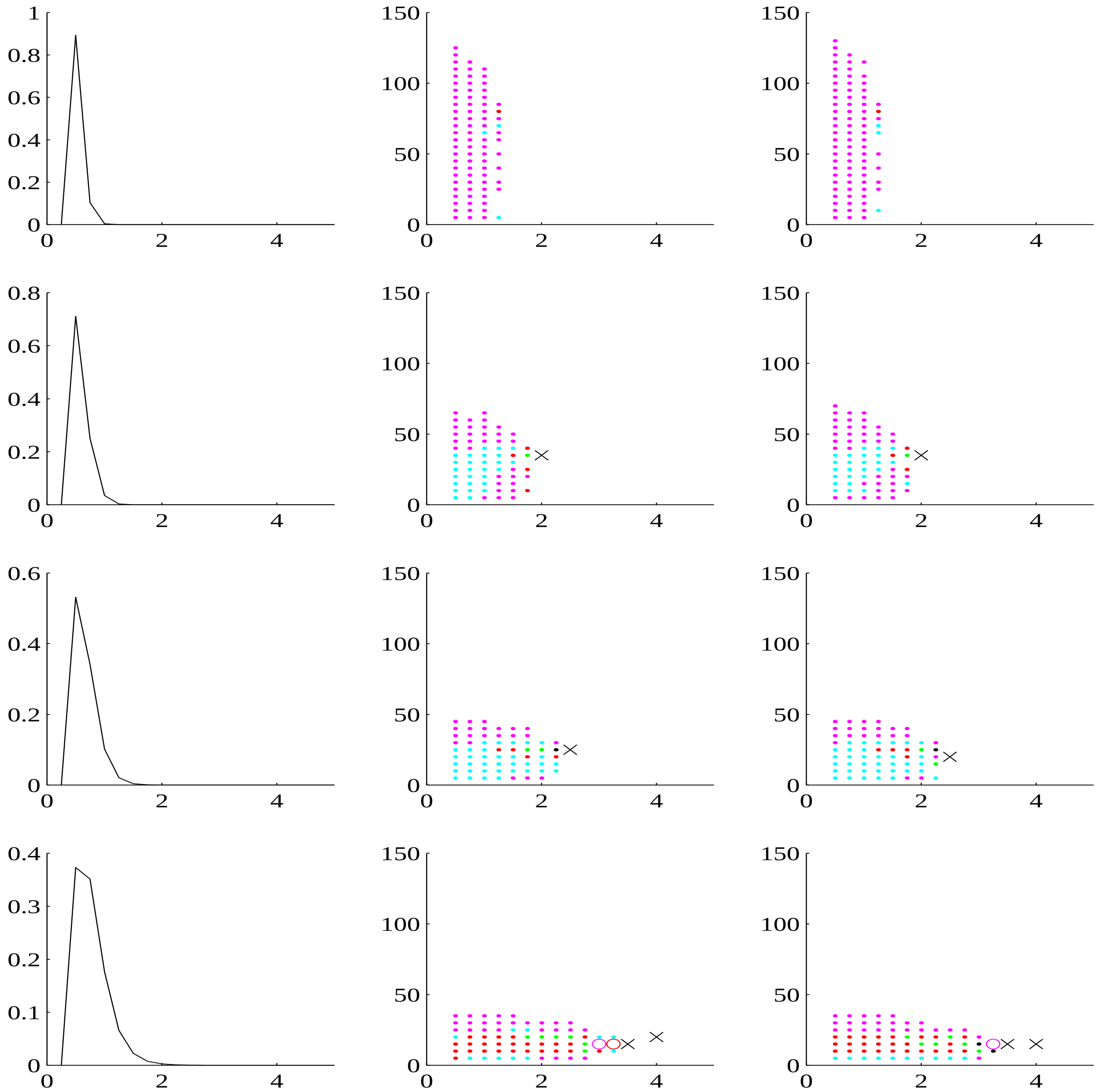


Figure A.33: Graphs show runs with 4 different loads and parameters (O = 1, I = 1, $t_a = 1$ s, $t_{rs} = .5$ s, $t_{ss} = .5$ s).

Table A.2: Input values for the second experimental design. The results are in Figures A.34 through A.39.

Figure	O	I	w_s	w_p	Figure	O	I	w_s	w_p
A.34	5	5	64	64	A.37	1	5	64	64
A.35	5	5	512	64	A.38	1	5	512	64
A.36	5	5	512	512	A.39	1	5	512	512

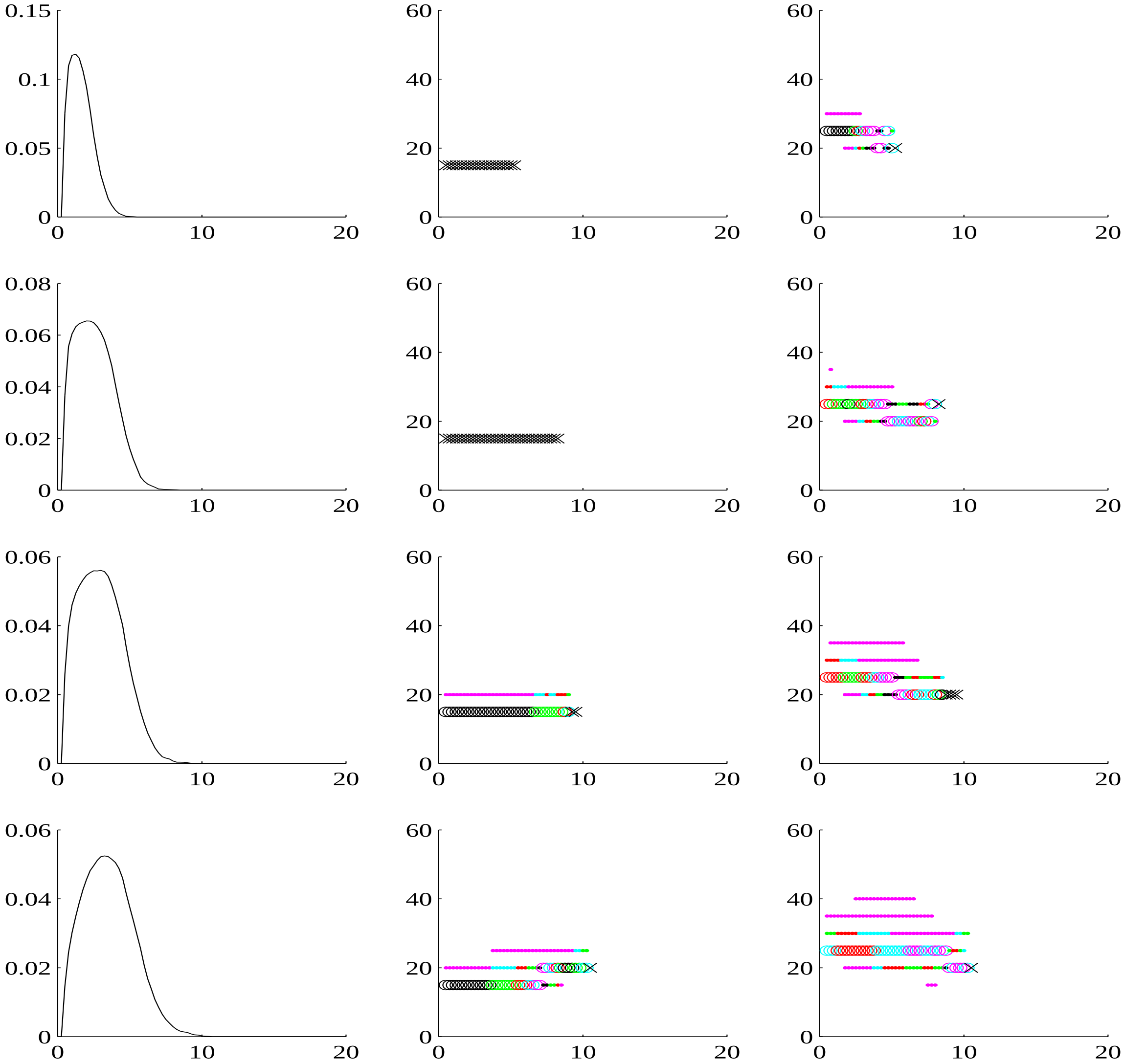


Figure A.34: Graphs show runs with 4 different loads and parameters (O = 5, I = 5, w_s = 64, w_p = 64).

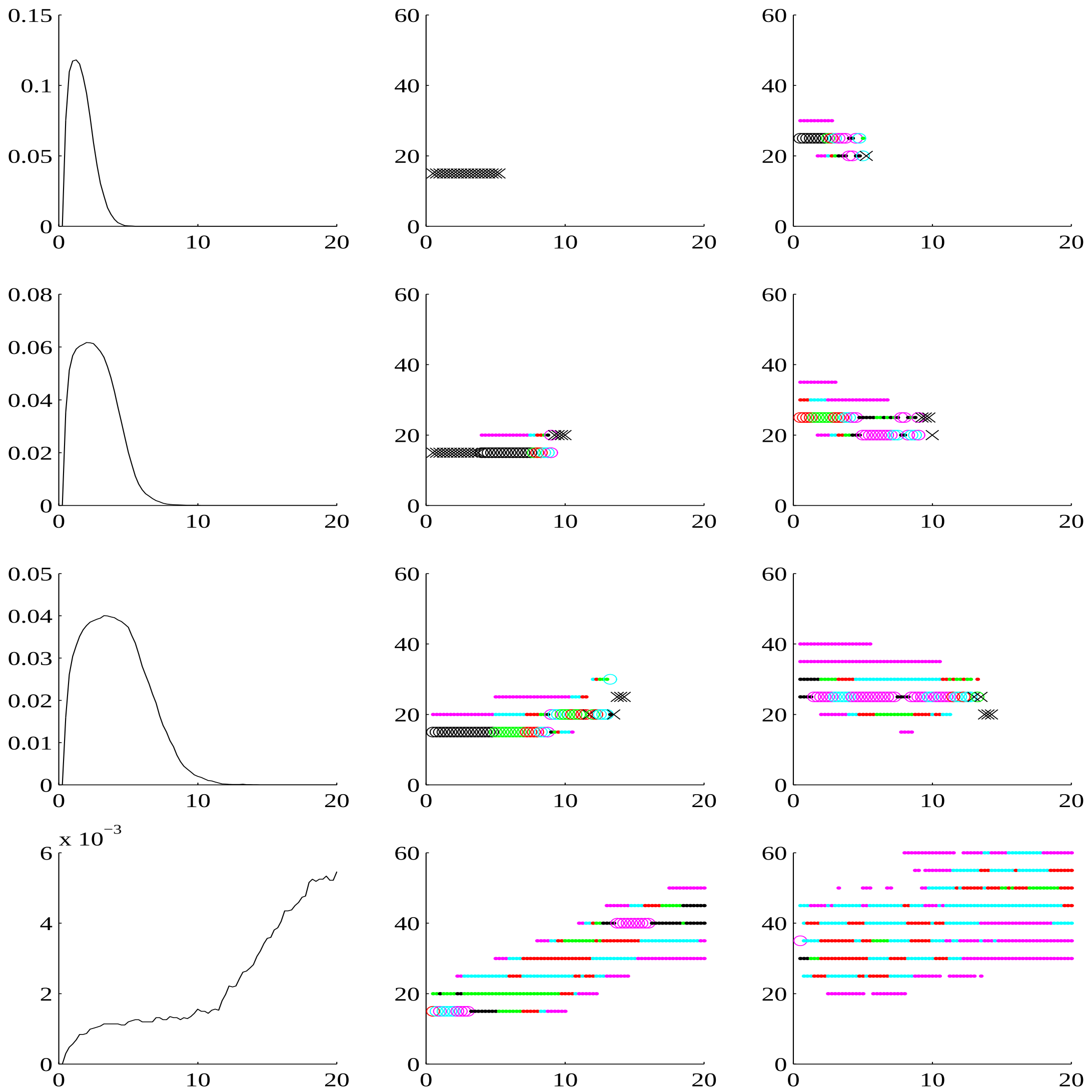


Figure A.35: Graphs show runs with 4 different loads and parameters (O = 5, I = 5, w_s = 512, w_p = 64).

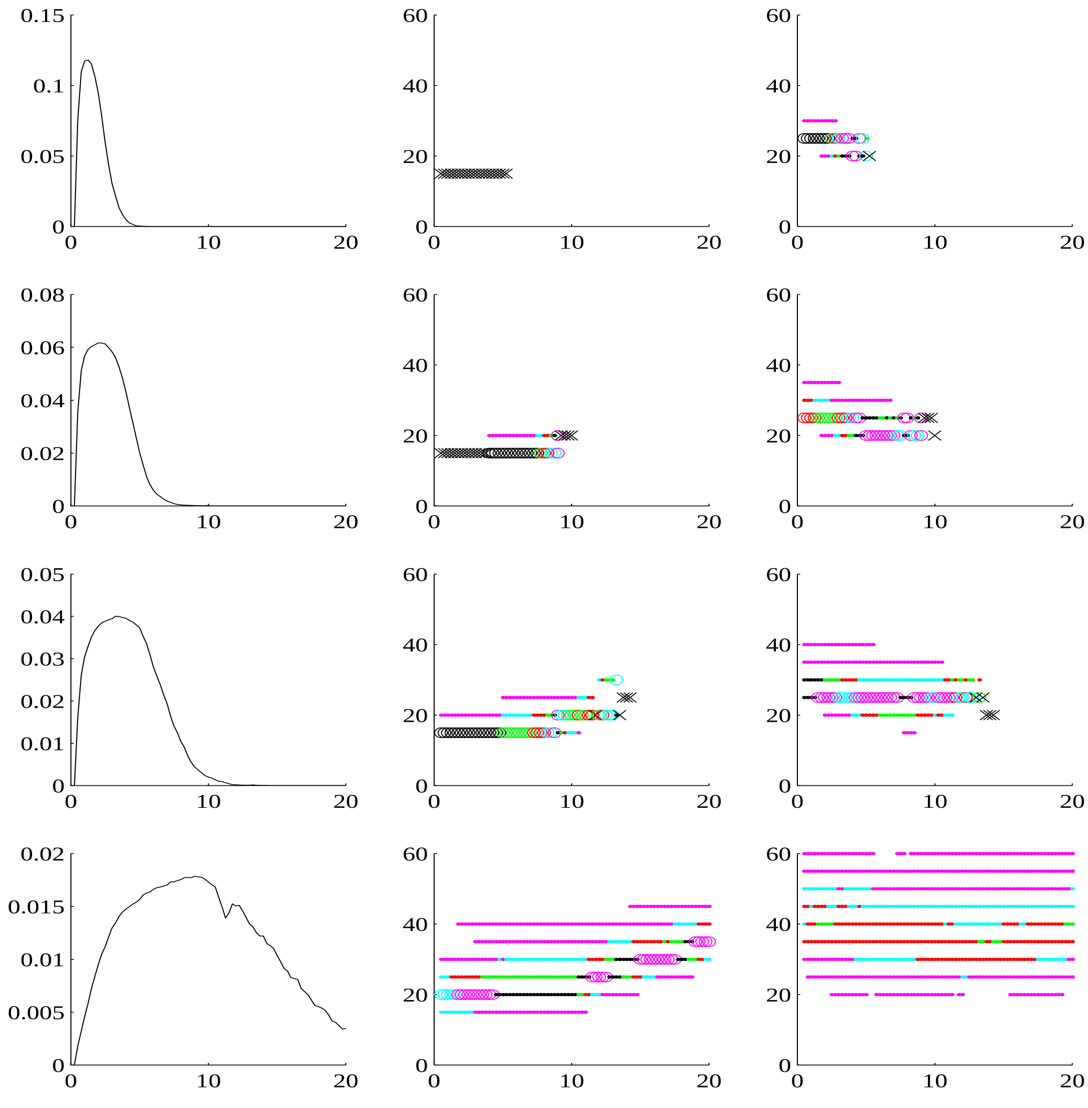


Figure A.36: Graphs show runs with 4 different loads and parameters (O = 5, I = 5, w_s = 512, w_p = 512).

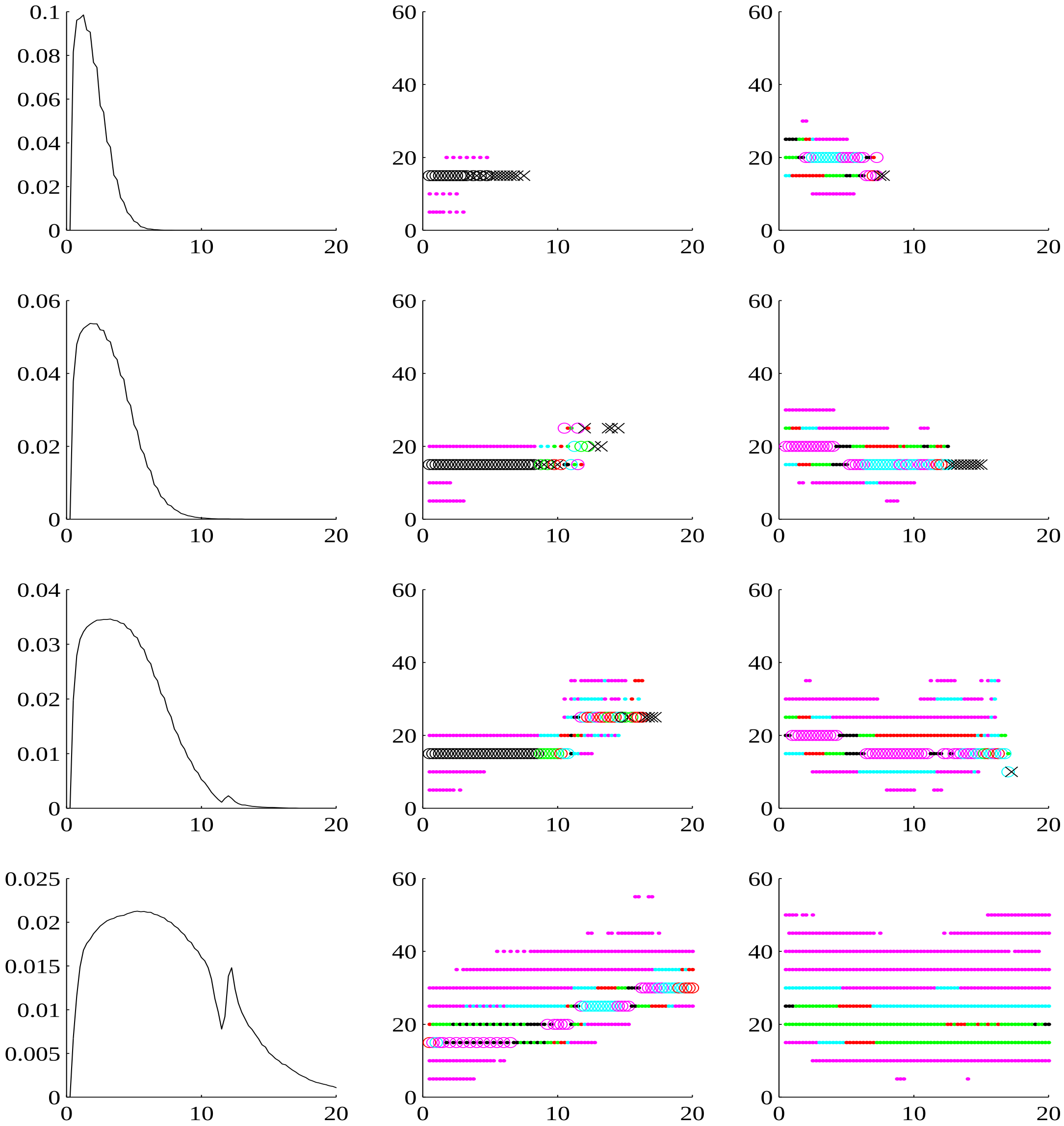


Figure A.37: Graphs show runs with 4 different loads and parameters (O = 1, I = 5, w_s = 64, w_p = 64).

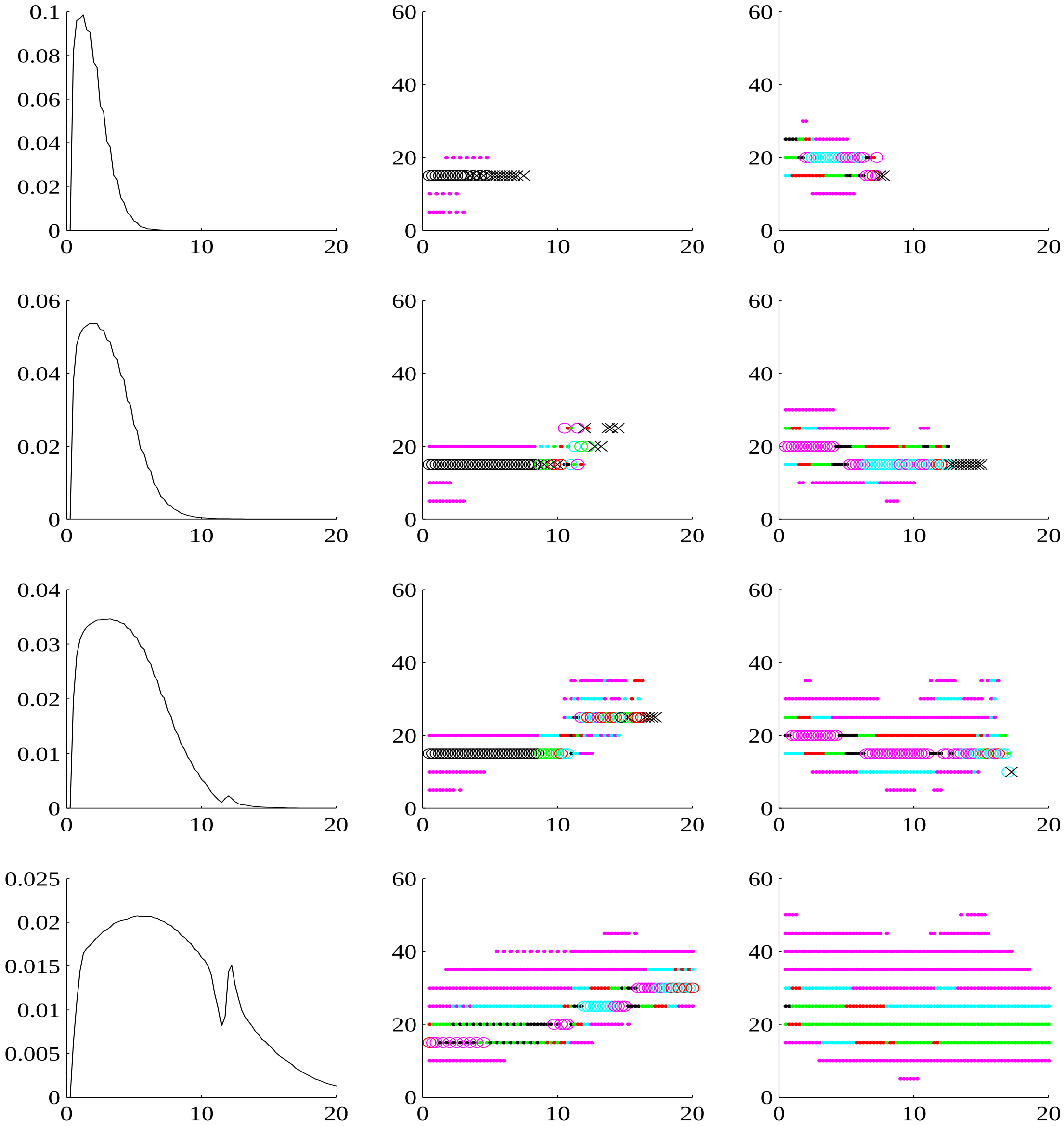


Figure A.38: Graphs show runs with 4 different loads and parameters (O = 1, I = 5, w_s = 512, w_p = 64).

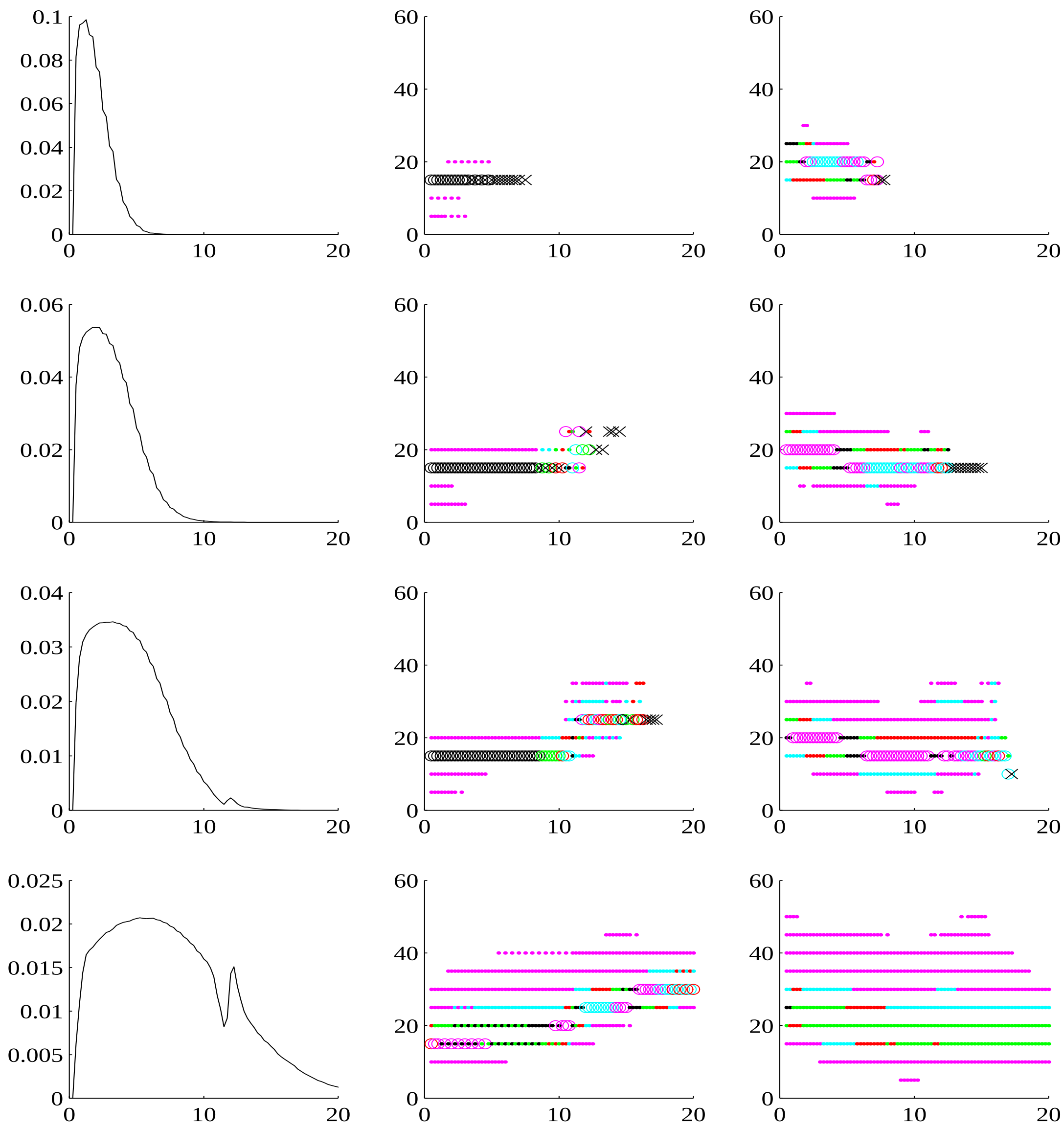


Figure A.39: Graphs show runs with 4 different loads and parameters (O = 1, I = 5, w_s = 512, w_p = 512).

Bibliography

[ADKM92a] Yair Amir, Danny Dolev, Shlomo Kramer, and Dalia Malki. Membership algorithms for multicast communication groups. In *Proceedings of the* 6^{th} *International Worshop, WDAG '92, Haifa, Isael*, November 1992. LNCS 647.

[ADKM92b] Yair Amir, Danny Dolev, Shlomo Kramer, and Dalia Malki. Transis: A communication sub-system for high availability. In *Proceedings of the* 22^{nd} *Conference on Fault Tolerant Computer Systems*, 1992.

[AMMS+93] Y. Amir, L.E. Moser, P. M. Melliar-Smith, D. A. Agarwal, and P. Ciarfella. Fast message ordering and membership using a logical token-passing ring. In *Proceedings of the Thirteenth International Conference on Distributed Computing Systems*, pages 551–560. IEEE, May 1993.

[BALL91] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. User-level interprocess communication for shared memory multiprocessors. *ACM Transactions on Computer Systems*, 9(2):175–198, May 1991.

[BBVv95] Anindya Basu, Vineet Buch, Werner Vogels, and Throsten von Eicken. U-Net: A user-level network interface for parallel and distributed computing. 1995.

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

[BC94] Kenneth P. Birman and Timothy Clark. Performance of the ISIS distributed computing toolkit. Technical Report TR-1432, Cornell University (Department of Computer Science), June 1994.

[BCG91] Kenneth P. Birman, Robert Cooper, and Barry Gleeson. Design alternatives for process group membership and multicast. Technical

Report TR-1257, Cornell University (Department of Computer Science), December 1991.

[BDGB95] Özalp Babaoğlu, Renzo Davoli, Luigi-Alberto Giachini, and Mary Baker. RELACS: A communications infrastructure for constructing reliable applications in large-scale distributed systems. Technical Report UBLCS-94-15, University of Bologna (Laboratory for Computer Science), January 1995.

[BGMP79] Mike Blasgen, Jim Gray, Mike Mitoma, and Tom Price. The convoy phenomenon. *Operating Systems Review*, 13(2):20–25, April 1979.

[Bir93] Kenneth P. Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12):37–53, December 1993.

[Bir94] Ken Birman. A response to Cheriton and Skeen's criticism of causal and totally ordered communication. *Operating Systems Review*, 28(1):11–21, January 1994.

[BJ87a] K. P. Birman and T. Joseph. Exploiting virtual synchrony in distributed systems. In *Proceedings of the Eleventh Symposium on Operating System Principles*, pages 123–138, November 1987.

[BJ87b] K. P. Birman and T. Joseph. Reliable communication in an unreliable environment. *ACM Transactions on Computer Systems*, 5(1):47–76, February 1987.

[BS95] Özalp Babaoğlu and André Schiper. On group communication in large-scale distributed systems. *Operating Systems Review*, 29(1):62–67, January 1995.

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

[Bv94] Kenneth P. Birman and Robbert van Renesse. *ISIS*. IEEE Press, 1994.

[CB91] Bernadette Charron-Bost. Concerning the size of logical clocks in distibuted systems. *Information Processing Letters*, 39(1):11–16, July 1991.

[CCI92] CCITT. *X.500 Directory Services*, 1992.

[Coo94] Robert Cooper. Experience with causally and totally ordered communication support: A cautionary tale. *Operating Systems Review*, 28(1):28–31, January 1994.

[CS93] David R. Cheriton and Dale Skeen. Understanding the limitations of causally and totally ordered communication. *Operating Systems Review*, 27(5):44–57, December 1993. Proceedings of the Fourteenth ACM Symposium on Operating System Principles.

[CS94] David R. Cheriton and Dale Skeen. Comments on the responses by Birman, Van Renesse and cooper. *Operating Systems Review*, 28(1):32, January 1994.

[CZ85] David R. Cheriton and Willy Zwaenepoel. Distributed process groups in the V kernel. *ACM Transactions on Computer Systems*, 3(2):77–107, May 1985.

[DB91] Himanshu B. Desai and John Bowles. Simulation and performance analysis of reliable broadcast protocols for distributed systems. In *Proceedings of the Twenty-Third Southeastern Symposium on System Theory, March 10-12, 1991*, pages 508–512, 1991.

[DC90] Stephen E. Deering and David E. Cheriton. Multicast routing in datagram internetworks and extended lans. *ACM Transactions on Computer Systems*, 8(2):85–110, May 1990.

[Dee89] Stephen E. Deering. Host extensions for ip multicasting. (RFC 1112), August 1989.

[Des91] Himanshu B. Desai. Simulation and performance analysis of reliable broadcast protocols for distributed systems. Masters dissertation, University of South Carolina, 1991.

[DM95] Danny Dolev and Dalia Malki. The design of the transis system. To appear in the proceedings of the Dagstuhl Workshop in Unifying Theory and Practice in Distributed Computing, September 1995, 1995.

[DMS94] Danny Dolev, Dalia Malki, and Ray Strong. An asynchronous membership protocol that tolerates partitions. Technical Report CS94-6, The Hewbrew University of Jerusalem (Institute of Computer Science), March 1994.

[dP93a] Martin de Prycker. *Asynchronous Transfer Mode: Solution for Broadband ISDN*. Ellis Horwood, 2 edition, 1993.

[DP93b] Peter Druschel and Larry L. Peterson. Fbufs: A high-bandwidth cross domain transfer facility. *Operating Systems Review*, 27(5):189–202, December 1993. Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles.

[EMS95] Paul D. Ezhilchelvan, Raimundo A. Macêdo, and Santosh K. Shrivastava. Newtop: A fault-tolerant group communication protocol. In *Proceedings of the Fifteenth International Conference on Distributed Computing Systems*. IEEE, May 1995.

[Fv95] Roy Friedman and Robbert van Renesse. Strong and weak virtual synchrony in horus. Technical Report TR95-1491, Cornell University (Department of Computer Science), March 1995.

[Gla95] Bradford Glade. *Publish/Subscribe Programming*. Ph.D. dissertation, Cornell University, In preparation 1995.

[GR93] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, 1993.

[GT90] Ajei Gopal and Sam Toueg. On the specification of broadcast. In *Proceedings of the Second IEEE International Workshop on Future Trends in Distributed Computing Systems*, pages 54–56, October 1990.

[Hey90] Andrew Heybey. The network simulator. Obtained via anonymous ftp from ath@ptt.lcs.mit.edu, September 1990.

[HHS94] Rainer Händel, Manfred N. Huber, and Stefan Schröder. *ATM Networks: Concepts, Protocols, Applications*. Addison-Wesley, 2 edition, 1994.

[IEE85a] IEEE. *IEEE 802.3: Carrier Sense Multiple Access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications*, 1985.

[IEE85b] IEEE. *IEEE 802.5: Token Ring Access Method and Physical Layer Specification*, 1985.

[ISO89] ISO. *ISO 9314: Information Processing System — Fiber Distributed Data Interface (FDDI)*, 1989.

[Jai94] Neel Jain. Personal communication, 1994.

[KTHB89] M. Frans Kaashoek, Andrew S. Tanenbaum, Susan Flynn Hummel, and Henri E. Bal. An efficient reliable broadcast protocol. *Operating Systems Review*, 23(4):5–19, October 1989.

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

[LK82] Averill M. Law and W. David Kelton. *Simulation Modeling and Analysis*. McGraw-Hill, Inc., 1982.

[LLSG92] Rivka Ladin, Barbara Liskov, Liuba Shrira, and Sanjay Ghemawat. Providing high availability using lazy replication. *ACM Transactions on Computer Systems*, 10(4):360–391, November 1992.

[MAMSA94] L.E. Moser, Y. Amir, P. M. Melliar-Smith, and D. A. Agarwal. Extended virtual synchrony. In *Proceedings of the Fourteenth International Conference on Distributed Computing Systems*, pages 56–65. IEEE, May 1994.

[Mar88] David Martin. Network simulator user's manual. Obtained via anonymous ftp from ath@ptt.lcs.mit.edu, September 1988.

[May92] Erwin Mayer. An evaluation framework for multicast ordering protocols. *Commputer Communications Review*, 22(4):177–187, October 1992. Proceedings of SIGCOMM '92.

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

[MMS94] L.E. Moser and P. M. Melliar-Smith. Probabilistic bounds on message delivery for the totem single-ring protocol. In *Real-Time System Symposium*, pages 238–248. IEEE, July 1994.

[MMSA91] L.E. Moser, P. M. Melliar-Smith, and Vivek Agrawala. Membership algorithms for asynchronous distributed systems. In *Proceedings of the Eleventh International Conference on Distributed Computing Systems*, pages 480–488. IEEE, May 1991.

[Mog92] Jeffery C. Mogul. Network locality at the scale of processes. *ACM Transactions on Computer Systems*, 10(2):81–109, May 1992.

[MPS91] Shivakant Mishra, Larry L. Peterson, and Richard D. Schlichting. A membership protocol based on partial order. In *Proceedings of the Second International Working Conference on Dependable Computing for Critical Applications*, February 1991.

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

[RB91] Aleta Ricciardi and Kenneth Birman. Using process groups to implement failure detection in asynchronous systems. In *Proceedings of the Tenth Annual ACM Symposium on Principles of Distributed Computing, August 19-21*, pages 341–353. ACM, 1991.

[Ric93] Aleta Ricciardi. *The Group Membership Problem in Asynchronous Systems*. Ph.D. dissertation, Cornell University, 1993. Also available as Cornell Computer Science Technical Report TR92-1313.

[Ros95] Marçel Rosu. Personal communication, 1995.

[Sch92] Lee W. Schruben. *Sigma: Graphical Simulation Modeling*. The Scientific Press, 1992.

[SF94] Marc Shapiro and Paulo Ferreira. Larchant-RDOSS a distributed shared persistent memory and its garbage collector. Technical Report TR94-1466, Cornell University (Department of Computer Science), November 1994.

[Shr94] Santosh K. Shrivastava. To CATOCS or not to CATOCS, that is the .. *Operating Systems Review*, 28(4):11–14, October 1994.

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

[Ste91] Patrick Stephenson. *Fast Ordered Multicasts*. Ph.D. dissertation, Cornell University, 1991. Also available as Cornell Computer Science Technical Report TR91-1194.

[van92] Robbert van Renesse. Personal communication, 1992.

[van93] Robert van Renesse. Causal controversy at Le Mont St.-Michael. *Operating Systems Review*, 27(2):44–53, April 1993.

[van94] Robert van Renesse. Why bother with CATOCS? *Operating Systems Review*, 28(1):22–27, January 1994.

[Vay95] Alexey Vaysburg. Horus object tools layer. Private communication, 1995.

[vB95] Robbert van Renesse and Kenneth P. Birman. Protocol composition in horus. Technical Report TR95-1505, Cornell University (Department of Computer Science), March 1995.

[vHB94] Robbert van Renesse, Takako M. Hickey, and Kenneth P. Birman. Design and performance of horus: A lightweight group communications system. Technical Report TR-1442, Cornell University (Department of Computer Science), August 1994.

[WMK94] Brian Whetten, Todd Montgomery, and Simon Kaplan. A high performance totally ordered multicast protocol. Technical Report TR-94-069, University of California at Berkeley (International Computer Science Institute), 1994.