The Best-Effort Virtual-Time CSMA/CD Protocols with Run-Time Clairvoyancy Support

Sam K. Oh
Glenn H. MacEwen

February 22, 1993
External Technical Report
ISSN-0836-0227-
93-347

Department of Computing and Information Science
Queen's University
Kingston, Ontario K7L 3N6

Document prepared February 23, 1993
Copyright ©1992, Sam Oh and Glenn MacEwen

Abstract

Two *clairvoyant* virtual-time CSMA/CD protocols for dealing with time-constrained messages are described. A clairvoyant protocol has *a priori* knowledge about messages expected to be produced in the future, so that it can take into account such *future messages* when it makes a decision for transmission. In the absence of clairvoyancy, some urgent future messages may not be successfully transmitted when they are actually produced.

For a protocol to be clairvoyant, run-time support is necessary. In the proposed method, the run-time kernel at each node provides future messages' characteristics, including the *latest sending times*. Since the latest sending time of a message specifies the point in time by which transmission of the message must begin, each proposed clairvoyant protocol regards (hence uses) it as the worst-case time at which the message can be produced. In fact, it is usual in real-time systems that messages are produced earlier than their latest sending times because the tasks producing them have shorter execution times than their worst-case estimates. Consequently, the clairvoyancy of a protocol can be enhanced by monitoring the execution times of task segments at run-time. Run-time support for clairvoyancy, provided by the run-time kernel and monitoring programs, is also described.

Simulation results show that the proposed clairvoyant protocols yield better performance over two other virtual-time protocols with no clairvoyancy for a reasonable range of system loads. The percentage of message timing failures is used as the primary performance measure.

Keywords: Real-Time. Communication Protocols. Clairvoyancy.
Timing Failure. Task Segment Monitoring.

Contents

1 Introduction **1**

2 Related Work **3**
- 2.1 Real-Time Protocols . . . 3
- 2.2 Monitoring Methods . . . 4
- 2.3 Run-Time Support for Clairvoyancy . . . 4

3 System Model **5**
- 3.1 Node Architecture . . . 5
- 3.2 Message Characteristics . . . 5
- 3.3 Message Transmission Model . . . 7

4 Best-Effort Protocols **7**
- 4.1 Problems Addressed . . . 7
- 4.2 Protocol Descriptions . . . 8

5 Run-Time Clairvoyancy Support **10**
- 5.1 Semantic Information . . . 10
- 5.2 Run-Time Support . . . 11
- 5.3 Assisting Task Scheduling . . . 12

6 Performance Analysis **13**
- 6.1 Simulation Model . . . 13
- 6.2 Performance Measures . . . 14
- 6.3 Comparative Analysis . . . 14

7 Conclusion **16**

1 Introduction

Real-time computer systems are required to produce correct results not only in their values but also in the times at which the results are produced. To satisfy such *timeliness* requirements, the execution times of program units called *real-time tasks* are constrained. Each task *timing constraint* can be either a *delay*, a *deadline*, or both. A delay constraint on a task specifies a point in time before which the task must not start, while a deadline constraint specifies a point in time at or before which the task must complete. Tasks can be categorized as either *periodic tasks* or *aperiodic tasks*. While a periodic task is *activated* repetitively at fixed intervals of time, an aperiodic task is activated in response to asynchronous events. Task execution times are, in general, stochastic. One often considers estimated worst-case execution times in analysing a set of tasks for schedulability.

For increased performance and reliability, distributed architectures having multiple nodes interconnected by a communications network are increasingly being used for real-time systems. Because tasks running on different nodes interact via network message-passing, the timely delivery of messages is essential for meeting task timing constraints. Consequently, in addition to task timing constraints, message delivery times must also be constrained.

Message delivery times can be constrained in a way similar to task execution times, by either a delay, a deadline, or both. A delay constraint on a message specifies a point in time before which the delivery must not commence, and a deadline specifies a point in time at or before which the delivery must finish. Messages in general are of variable length, so different durations are required for transmission. These *transmission times*, the total time from start of sending to completion of receiving, must be taken into account for successful delivery of messages. The *latest sending time* of a message is the point in time at or before which transmission must begin (i.e. deadline - transmission time). The *earliest sending time* of a message is the point in time before which transmission must not begin (i.e. delay - network traversal time).

A class of networks frequently used for real-time applications is the *multiple access network* in which nodes, and hence tasks, share a single communication channel. In this class of network, only a single message can be in transit over the channel at any one time. If two or more messages are simultaneously transmitted on the channel, none will be correctly delivered to their destination due to a *collision*. Therefore, such systems must have synchronization *protocols* for controlling the transmission of messages [PZ78]. Based on how collision is handled, protocols for multiple access networks can be categorized into *controlled-access* and *contention-based* [KSY84]. The controlled-access protocols are characterized by *collision-free* access to the channel; they allow each node to access the channel without collision. Time division multiple access (TDMA) and token passing protocols are examples. On the other hand, contention-based protocols allow nodes to transmit messages simultaneously over the channel; hence collisions can occur. Various carrier-sense-multiple-access protocols (CSMA) with or without collision detection (CD) belong to this category. The performance of the multiple access protocols in these classes may vary according to their application area. Many of them, however, are not intended for real-time systems. We call protocols devised to deal with time-constrained messages *real-time communication protocols* (or simply real-time protocols). In this paper, we focus on real-time protocols for transmitting messages with deadline constraints.

It is known from task scheduling theory that the *earliest deadline first* (EDF) and the *least laxity first* (LLF) scheduling algorithms are *optimal* [1] in a uniprocessor system in which the configuration is *static*, i.e. all the task characteristics are known *a priori*. In practice,

[1] An optimal scheduler is one which may fail to meet a deadline only if no other scheduler is guaranteed to succeed [DM89].

there are many *dynamic* applications in which sufficient knowledge regarding the task characteristics cannot be obtained *a priori*. Even in such dynamic environments, the EDF and LLF policies have been shown to yield better performance than others [JLT85]. There is an analogy between multiple access real-time protocols and task scheduling algorithms for uniprocessors because, in both cases, a single resource is being allocated. Based on this analogy, several real-time protocols adopt a policy that approximates the EDF or the LLF policy [KSY83, PTW88, ZR87, ZSR90]. We call a message that must be transmitted first the *most urgent message*. A message with the least laxity becomes the most urgent one in a protocol using the LLF policy, while a message with the earliest deadline does so in a protocol using the EDF policy.

Clairvoyancy is the ability to possess *a priori* knowledge about the characteristics of future events. In real-time systems, a message expected to be produced in the future can be more urgent than those already available for transmission. Such an urgent *future message* may not be delivered by its deadline if less urgent but available messages are transmitted first. In addition, a message transmitted too early may result in a collision with a more urgent message from another node. If a communication protocol has clairvoyancy with respect to future messages (so that it can consider them when it makes a decision for transmission) such undesirable and even harmful situations may be avoided or minimized. Recently, several real-time CSMA protocols have been introduced in the literature [KSY83, PTW88, ZR87, ZSR90]. However, none of these protocols have clairvoyancy.

In this paper, we introduce two clairvoyant real-time CSMA/CD protocols that make use of the concept of the virtual clock [MK85, ZR87]; one adopts the LLF policy and the other the EDF policy. In the sense that these two protocols attempt to approximate a basic policy as closely as possible and to transmit as many messages as possible successfully, we call them *best-effort protocols* and denote the former by *BVTL* and the latter by *BVTD*. These two protocols attempt to reserve the communication channel for a most urgent future message if it cannot be successfully transmitted when a less urgent but currently available message is transmitted first.

The best-effort protocols require run-time support for clairvoyancy. In our method, the run-time system at each node provides the characteristics of a future message (i.e. design-time information such as deadline and length) when the message is *recognized* to be produced in the future. The best-effort protocols, with this *basic clairvoyancy* support, take the latest sending time of a future message as the worst-case time at which the message can be produced. In fact, it is usual in real-time systems that a message is produced earlier than its latest sending time because the actual execution time of its producer task may be less than the estimated worst-case execution time. With run-time monitoring of task segments' execution times, the best-effort protocols can have *enhanced clairvoyancy*, i.e. with more accurate run-time information about when future messages are to be produced, so that they can have better opportunities for transmitting more urgent future messages. Run-time clairvoyancy support by the run-time system and monitoring programs is also described in this paper.

The remainer of this paper is organized as follows. Section 2 briefly surveys related work. Section 3 defines the system model, which consists of a set of nodes interconnected by a single communication channel; each node is a multiprocessor system with shared memory. We assume a dynamic system environment in which a task arrives asynchronously at a node; by *task arrival* we mean the occurrence of a request to execute the task. For the case in which task execution is requested by some sequence of requests, the task arrival is the occurrence of the last request in the sequence. Section 4 introduces the best-effort protocols. Section 5 describes run-time support for clairvoyancy. Section 6 presents our simulation results, and Section 7 gives conclusions.

2 Related Work

2.1 Real-Time Protocols

Several real-time protocols are based on EDF or LLF [KSY83, PTW88, ZR87, ZSR90]. Kurose *et al* [KSY83] describe a protocol in which each message is required to be sent within a fixed time interval, K time units from its production time. Since messages produced earlier will always be more urgent than those produced later, a message that arrives first will be transmitted first. Panwar *et al* [PTW88] assume that a message is constrained by a time interval that is a random variable with general distribution. Also, message lengths are taken to be a sequence of independent and identically distributed random variables. With these assumptions, the lengths of messages vary with the order in which they are transmitted. However, in real-time systems messages produced later may be more urgent than those produced earlier, and message lengths, being often fixed at design-time, are invariant with the order in which they are transmitted. For these reasons, we exclude these two protocols from our discussion.

In conventional window-based CSMA/CD protocols, a *window* defines an interval on the axis of some message parameter (e.g. production time, deadline, latest sending time). When a collision occurs, the window is split into halves, and each node treats messages in the left half first and in the right half next. When a new message is produced in the left half while the right half is being treated, the message is not considered for transmission. Zhao *et al* [ZSR90] introduce a new window protocol *WL* in which a window is formed on the latest sending time axis, and messages are queued at each node according to their latest sending times. The deadline of a message in this protocol, rather than being relative to its production time, is specified as a point in time by which it must be delivered. Also, messages have fixed lengths which are invariant with the order in which they are transmitted, and are allowed to have arbitrary laxities. One major difference between this protocol and conventional window-based protocols is that a newly produced message can be considered for transmission by letting the window lower-bound be the current time.

Molle and Kleinrock introduce a virtual-time CSMA/CD protocol (VTCSMA) [MK85] in which a virtual clock at each node runs along the arrival time axis. In this protocol, each node maintains two clocks: a real clock and a virtual clock. The real clock runs at unit speed, while the virtual clock runs at a higher rate. Zhao and Ramamritham [ZR87] generalize this protocol by allowing four different message parameters, transmission time, arrival time, latest sending time, and deadline, to be associated with the axis along which a virtual clock runs. We name these protocols *VTT*, *VTA*, *VTL*, and *VTD* respectively, using the obvious acronyms. Among these four protocols, the *VTL* and *VTD* protocols have been shown to be suitable and hence recommended for real-time systems [ZR87]. These two protocols assume the same message characteristics as those in *WL*. Messages are queued according to the increasing order of their latest sending times in *VTL* and of their deadlines in *VTD*. The VTL protocol can be outlined as follows (The VTD protocol is shown in parentheses.):

- When a message is produced, a virtual time parameter VS is initialized to its latest sending time (deadline in *VTD*).
- When the channel has been continuously idle a node finds the most urgent message, for which $VS \leq vt$, and transmits it. A virtual time vt denotes a value read from a virtual clock.
- When the channel is sensed busy, the virtual clock does not run.

- The channel will become idle either after a successful transmission or a collision. All tardy messages are discarded from the queue. The virtual clock is reset to the value of the real clock rt. If a most urgent message whose $VS = vt$ is found, it is transmitted (Note that $vt = rt$.).
- When there is a collision the sender node retransmits the message with probability P, or it modifies VS for the message; a new VS value is drawn from the interval (rt, ls) ((rt, dl) in VTD) where ls denotes the latest sending time of the message (and dl denotes the deadline). Finally, the message is put back into the message queue.

2.2 Monitoring Methods

Monitoring techniques, long used for system testing, debugging and performance monitoring [Cas91, HW90, LSMC90, TFCB90, TKM88], have been applied to critical and/or real-time systems to detect system errors or property violations [Gor91, CJD91, JG90, MM88, WS88, Lu82]. Similar techniques have also been applied in real-time systems to aid scheduling of real-time tasks [GG90, HS90]; the basic idea is to enable scheduling algorithms to use underutilized system resources more efficiently and hence to improve performance. We survey two such techniques in the following paragraphs; both assume *independent non-interacting* tasks, i.e. no message communication.

Haban and Shin [HS90] propose a monitoring approach in which each application task with random execution times is divided into a set of disjoint program *segments* according to the syntactic structure of the task. Assuming knowledge of the maximum number of loops within each of these segments, the worst-case pure execution time for each segment is estimated. In their approach, a specially designed processor, called TMP, measures the true execution time and resource sharing delay of tasks at run-time. These measurements enable the run-time scheduler, a part of TMP, to schedule tasks adaptively.

Gopinath and Gupta [GG90] propose a 'compiler-assisted' approach in which the compiler examines the code of each application task and partitions it into *typed* segments. The type of each segment is determined by the combination of two criteria: predictability and monotonicity. A segment is predictable if it has a fixed execution time; otherwise, where the execution time is determined by input data, it is unpredictable. A segment is *monotonic* if its output quality is monotonically improved as it is executed longer; otherwise it is non-monotonic. During compilation, program segments are re-ordered so that segments are executed unpredictable before predictable, and non-monotonic before monotonic. At the end of each unpredictable segment, measurement code is inserted to measure the actual execution time of the segment; a time deviation is calculated by subtracting this measured time from the estimated worst-case execution time of the segment. If the accumulated segment time deviations of a task indicate a timing error, the run-time scheduler takes recovery action by changing the loop bounds of monotonic segments. Tasks that cannot be recovered are aborted.

2.3 Run-Time Support for Clairvoyancy

In a real-time distributed system with a dynamic environment, being clairvoyant may be very difficult for a task scheduler, but may be practical for a communication protocol. In the following sections, we first introduce two virtual-time CSMA/CD protocols that make use of run-time-provided clairvoyancy to improve performance. Second, run-time system support to provide the clairvoyancy is described. An application of a task monitoring technique for enhancing protocol clairvoyancy is also described.

3 System Model

Consider a distributed system of N nodes connected by a single communication channel. Each node is a multiprocessor system with shared memory. Resources in the shared memory include message queues and other shared information structures. Each processor in a node has its own local memory for task code and private resources, and communicates with the other processors in the same node via the shared memory. We assume a dynamic system environment; that is, aperiodic tasks arrive asynchronously at nodes. We also assume synchronized node clocks.

3.1 Node Architecture

Figure 1 shows the system architecture. Each node comprises an *application subsystem* (APS), an *adaptive control subsystem* (ACS), and a *communications subsystem* (COS). The APS consists of a set of *application processors* on which tasks (mostly application tasks) execute. The ACS and the COS are described in the following paragraphs.

The ACS comprises a *node scheduler*, a *dispatcher*, a *message queuer*, a *message distributor*, and *fault-tolerant and adaptive methods*. The node scheduler attempts to schedule (allocate to a processor) a newly arrived task so that it can be completed by its deadline while retaining previously scheduled tasks. The dispatcher invokes a next task for execution among those scheduled by the node scheduler. The message queuer queues produced messages for transmission. The message distributor delivers messages received via the COS to corresponding consumer tasks. There also exist methods for system fault tolerance, adaptiveness and performance improvement; these methods include monitoring algorithms and task migration algorithms. Each application processor is assigned a copy of the dispatcher, a copy of the message queuer, and some monitoring programs. A specially designed processor is dedicated to the remaining ACS functions. Since the application tasks are unaffected by system overheads such as external interrupts and task scheduling, their execution behavior is more predictable. Similar approaches are found in [HS91, HS90, SR87].

A *node feasible schedule* is one in which constraints on all tasks are guaranteed to be met. When a task arrives, if there is a feasible schedule the node scheduler accepts the task and assigns a *scheduled start time* and a *scheduled completion time*; such a task is said to be *activated*. When another task arrives, the node scheduler may reschedule the previously scheduled tasks in order to accept it. Accordingly, the scheduled start and completion times assigned to the previously scheduled tasks may be altered. The scheduled completion time assigned to a task is not greater than the task deadline, and the actual execution time of a task is assumed to be less than or equal to its worst-case estimate. To increase resource utilization, a task may start earlier than scheduled if the node scheduler and dispatcher are adaptive. Finally, a task may not be schedulable by the node scheduler; in such a case, it can be transferred to another node using some task node assignment protocol such as a bidding algorithm.

The COS controls the transmission and reception of messages, for which there exists a *message transmission protocol* and a *message reception protocol*. The former transmits messages produced by application and system tasks, while the latter accepts messages from the communication channel (CH) and passes them to the ACS (i.e. to the message distributor) if they are not corrupted. Corrupted messages are discarded. There is a dedicated controller for the COS.

3.2 Message Characteristics

A message m is characterized by the following parameters:

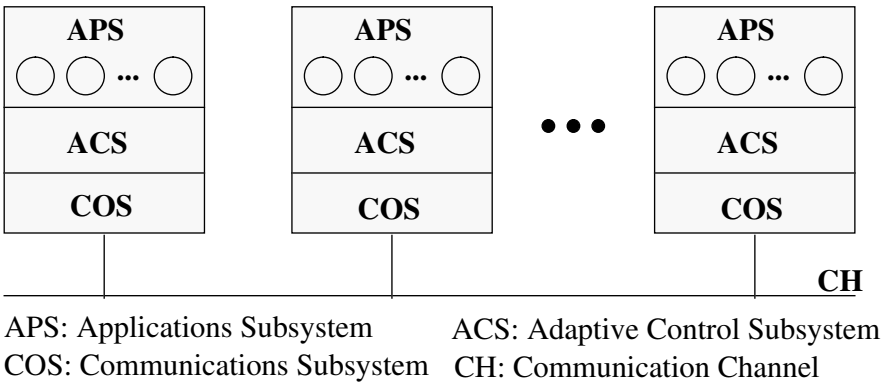

Figure 1: System Model

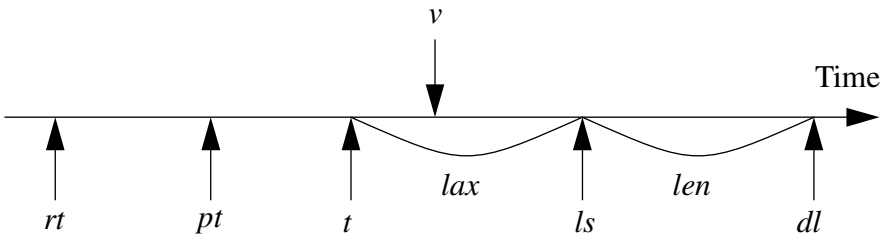

Figure 2: Various Times and Parameters Associated with A Message

- *pt(m)*: production time, the time at which m becomes available for transmission,
- *rt(m)*: recognition time, the time at which the production of m is recognized.
- *dl(m)*: deadline, the time by which m must have been delivered to its destination,
- *len(m)*: length, the total number of bits in m, i.e. the total number of time units needed to transmit the message without collision, and
- *ls(m)*: latest sending time, $dl(m) - len(m)$, i.e. the latest time for guaranteeing timely delivery. (Notice that this assumes a propagation delay of one time unit.)

The *real time* (sometimes called *current time*) is the value of a local physical clock, while the *virtual time* is the value of a virtual clock. We normally denote the former by t and the latter by v. The ratio of virtual to real clock speed is η. The laxity of a message m, *lax(m)*, is defined to be $ls(m) - t$. A message is said to be *tardy* when its laxity becomes negative. Unless specified otherwise, all times characterizing a message are real. When clear, we may omit the argument 'm' from the above message characteristics. Figure 2 illustrates the relationship among the various times and parameters associated with a message.

Since we focus on message transmission protocols, the tasks that do not produce messages are not our concern; hence, we deal with only the tasks that produce messages. We also assume that messages are recognized when their producer tasks are activated. Messages are also assumed to be produced and queued no later than their latest sending times.

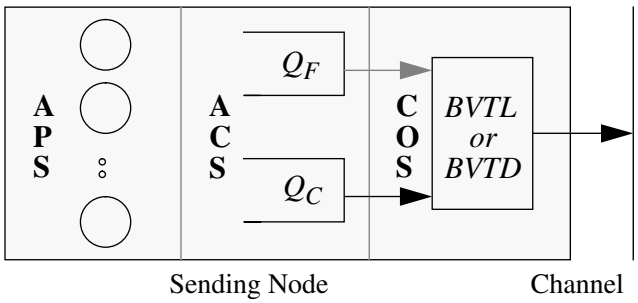

Figure 3: Node Architecture for Best-Effort Protocols

3.3 Message Transmission Model

CSMA/CD transmission protocols can operate in either an *asynchronous* (unslotted) or a *synchronous* (slotted) mode. In asynchronous mode, each node runs a copy of the protocol independently using local observations of the channel. In synchronous mode, all nodes run their local copies of the protocol in lock-step; the time axis is divided into a sequence of time units, called *slots*, and each node can transmit messages only at the beginning of each slot. We assume synchronous mode; that is, the best-effort protocol in each node is invoked at the beginning of each slot. The maximum end-to-end propagation delay for a bit is the slot length τ.

As shown in Figure 3, each node maintains two queues:

- a *future queue* Q_F containing the characteristics of future messages and
- a *current queue* Q_C containing the messages that have been produced and are not tardy.

Messages produced by application tasks are queued in the order of latest sending time with $BVTL$, and of deadline with $BVTD$. When a future message is recognized, the ACS places an entry into Q_F. When it is actually produced, the ACS moves the entry from Q_F to Q_C. Tardy messages are simply discarded from Q_C. While the solid arrows in Figure 3 represent message retrieval and transmission, the dotted arrow merely denotes information about the characteristics of future messages.

4 Best-Effort Protocols

4.1 Problems Addressed

Several problems can be identified in the VTL and VTD protocols. Although these problems may occur only rarely, depending upon the characteristics of the application tasks. the impact may be significant for particular applications.

- A collided message may become tardy. Since message tardiness is not checked at the time of a collision, a collided and tardy message can be retransmitted.
- At the time of a collision, there may be a message that is newly produced and the most urgent, but which is not considered for transmission.

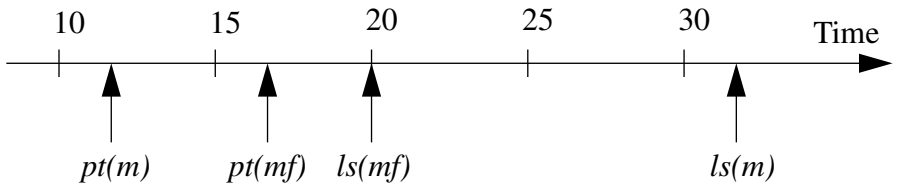

Figure 4: Message Production and Latest Sending Times

- Both the VTL protocol and the VTD protocol may waste one time-unit when the channel becomes idle after the successful transmission or the collision of a message. As a result, a message that could be successfully transmitted, can become tardy. For example, consider a message m with $ls = 300$ and $dl = 330$ and assume that the virtual clock runs at 35 times the rate of the real clock, i.e. $\eta = 35$. It collides at $t = 299$ and is queued back with modified VS (say, $VS = 299$ in VTL and $VS = 325$ in VTD). Assume an idle state at $t = 300$ (after the collision). In this state, each protocol discards tardy messages and sets v equal to t; if a message whose $VS = v$ is found, it is transmitted. Since m has $VS < v$ ($VS = 299$) in VTL and $VS > v$ ($VS = 325$) in VTD, it cannot be transmitted and becomes tardy in the next idle state (at $t = 301$). By letting the virtual clock advance η (one tick of the real clock) and modifying '$VS = v$' into '$VS \leq v$', m may be successfully transmitted. Also, message tardiness needs to be checked at the continuous channel idle state to prevent transmission of tardy messages.

In addition to these problems, the virtual clock may greatly exceed the real clock when the system is lightly loaded. As a consequence, a message may be transmitted unnecessarily early, which may in turn aggravate system performance. Consider the example shown in Figure 4. The channel is initially idle at $t = v = 0$. The virtual clock starts to run at three times the rate of the real clock ($\eta = 3$). Assume that message m with $ls = 32$ and $len = 10$ is produced at $t = 12$ ($v = 36$). Since $VS(m) < v$, the VTL protocol starts transmission of m; the virtual clock stops. The transmission of m completes at $t = 22$. Assume that another message mf with $ls = 20$ and $len = 5$ is produced during the transmission of m, say at $t = 17$. Since $ls(mf) = 20$, mf becomes tardy and is discarded. If the protocol has clairvoyancy about mf, it could transmit both m and mf successfully by delaying m's transmission until $t = 20$ (because $ls(mf) = 20$). A similar situation can occur in the VTD protocol.

4.2 Protocol Descriptions

In the best-effort protocols, as in the VTL and the VTD protocols, each node maintains two clocks: a virtual clock and a real clock. The virtual clock runs at a rate η times faster than that of the real clock. While the channel is busy, the virtual clock stops. When a message is newly produced, the virtual parameter VS is initialized to ls in $BVTL$ and to dl in $BVTD$. As shown in Table 1, there are five possible protocol states. The actions taken in each of these states are as follows:

- State II:
  The virtual clock runs continuously. Tardy messages are discarded. If there exists a most urgent message m in Q_C whose $VS \leq v$ there are three possibilities:

  C1: There exists no most urgent future message. m is transmitted.

Protocol State	Channel State at Previous Slot	Channel State at Current Slot
II	idle	idle
BI	busy	idle
CI	collision	idle
BSY		busy
COL		collision

Any Allowed Channel State

Table 1: Protocol States

C2: There exists a most urgent future message *mf*, and *mf* can be delivered by its deadline even after m's transmission (i.e. $lax(mf) > len(m)$). m is transmitted.

C3: There exists a most urgent future message *mf* which cannot be successfully transmitted. In this case the algorithm checks whether or not m could be successfully transmitted after waiting for and transmitting *mf*. There are two subcases:

C3a: m could be transmitted later (i.e. $lax(m) > lax(mf) + len(mf)$). In this case the algorithm waits until *mf* is produced. (We discuss two waiting strategies later in this section.)

C3b: m could not be transmitted later. m is transmitted.

- States BI and CI:
  The virtual clock is set equal to $t + \eta$. Tardy messages are discarded. If there is a most urgent message m whose $VS \leq v$, there are four cases corresponding to cases C1, C2, C3a, or C3b.

- State BSY:
  A node that is transmitting a message continues its transmission. Other nodes wait for the completion of this transmission.

- State COL:
  If a collided message becomes tardy, it is discarded. Otherwise, each node checks whether there exists a newly produced most urgent message. If so, the collided message is put back into the message queue Q_C. The associated VS parameter is modified by a value randomly drawn from (t, ls) in $BVTL$ and from (t, dl) in $BVTD$.

  With the most urgent message m (it could be a collided message or a newly produced message), check whether or not there exists a future message that will be the most urgent when produced. There are four cases corresponding to cases C1, C2, C3a, or C3b. In cases C1, C2 and C3b, m is either transmitted with retransmission probability P, or put back into the message queue Q_C. In case C3a, m is put back into the queue, and a suitable waiting action is taken (See below.).

Two waiting strategies can be used for the most urgent future message *mf*:

- a *local waiting strategy* and

- a *global waiting strategy.*

With a local waiting strategy, only its host node waits for a message *mf*, while all the nodes wait in the global waiting strategy. With local waiting, the host node simply waits for *mf* without sending any messages. With global waiting, the host node transmits a *synchronization message* with the shortest length (i.e. one-bit message). When the channel is sensed idle after the successful transmission or collision of the synchronization message, each node resets its virtual clock to $t + \eta$. As a result, the most urgent future message has a chance for transmission. The *ls* value of a synchronization message is taken to be equal to r. Consequently, when it has a collision, it becomes tardy and is discarded. The best-effort protocols use local waiting only when *lax(mf)* is one. Otherwise, global waiting is used.

5 Run-Time Clairvoyancy Support

Monitoring techniques generally involve two phases:

- a *setup phase*, and
- a *checking phase.*

During the setup phase, usually at design time, a monitoring system or program is provided with semantic information about the program to be monitored. Such semantic information may include an allowed range for a monitored value or condition, an allowed sequence of event occurrences, a time-bound within which a specified event must occur, or an estimated execution time of a real-time task. During the checking phase at run-time, the monitoring system collects run-time information about the program being monitored and observes any discrepancies between this information and the design information. A discrepancy may indicate a violation of a system property, or provide information about resource utilization.

The best-effort protocols attempt to wait when a most urgent future message could not be successfully transmitted if a less urgent but currently available message is transmitted first. However, no waiting occurs if the less urgent message would become tardy as a result. The value *lax(mf)* occurring in case 'C3a' of the best-effort protocols represents the time interval that a best-effort protocol must wait for *mf* in the worst-case. Since a message is usually produced earlier than its latest sending time, it is possible to increase the chance to transmit a more urgent future message if the protocol is provided with more accurate information about when future messages will be produced. A monitoring technique can be used to provide such information; the monitoring program predicts the production time of a message by measuring the segment execution times of the task producing the message. We call this the *measured production time* of a future message *mf* and denote it by *mpt(mf).* To accommodate this monitoring support, we modify case 'C3a' in the protocol description as follows:

$$lax(m) > rtup(mf) + len(mf)$$

where *rtup(mf)* (read as 'remaining time until production' of *mf*) is defined to be $mpt(mf) - t$.

5.1 Semantic Information

An aperiodic task execution T is characterized by the following parameters:

- *ar(T)*: *arrival time*, the point in time at which T arrives.
- *dl(T)*: *deadline*, the point in time by which T must complete,
- *wet(T)*: *worst-case execution time*, the estimated time interval needed to complete T in the worst-case,

- *PL(T)*: *placement constraint*, a specification of a particular processor or processors on which T must be assigned for execution, and
- *R(T)*: *resource requirement*, the set of resources required by T.

When a task arrives, it is activated by the node scheduler if there is a feasible schedule. The *activation time* of T is denoted by *at(T)*. All the required resources for a task execution are allocated when the task is activated, and are not released until the completion or cancellation of the task [2]. Tasks having resource conflicts cannot be scheduled in parallel. Once started, tasks cannot be preempted. The *start time* and the *completion time* of a task T are denoted by *st(T)* and *ct(T)* respectively. We define the *start delay* of a task T to be $st(T) - at(T)$, which represents the elapsed period of time from T's activation to T's start; it is denoted by *std(T)*. A task may produce multiple messages. Of course, each task will not produce its first message earlier than its start time nor produce its last message later than its completion time (hence its deadline).

The semantic information about a task is obtained by dividing it into a set of *sequential disjoint* program segments based on its syntactic structure. A statement in which a message is produced becomes a natural point for task segmentation. The information needed for estimating worst-case segment execution time includes the longest execution paths, the maximum number of loops, and the worst-case message queuing delays and resource holding times. We denote the estimated worst-case execution time of a segment S_i of a task T by $wet(T.S_i)$. A task T's estimated worst-case execution time is the sum of its segments' worst-case execution times:

$$wet(T) = \sum_{i=1}^{n} wet(T.S_i)$$

We also specify for each segment the set of required resources. A *resource set* required for a segment S_i of a task T is denoted by $R(T.S_i)$. The set of required resources is the union of its segments' resource sets:

$$R(T) = \bigcup_{i=1}^{n} R(T.S_i)$$

The information about segmented tasks and the characteristics of tasks and messages form the semantic information base for monitoring. Figure 5 shows an example of a segmented task T; it consists of n sequential program segments and produces two messages *m1* and *m2* at the end of segment S_j and of segment S_n. Note that a message is recognized when its producer task is activated.

5.2 Run-Time Support

There are two ways to monitor real-time task behavior:

- an *embedded method*, and
- a *concurrent method*.

With the embedded method, a monitoring program is inserted into the program being monitored. With the concurrent method, a monitoring program runs in parallel with a program being monitored; a program statement, which invokes a routine that provides run-time information to the monitoring program, is inserted at particular points in the program being monitored. For the monitoring of task segments, we use the embedded method.

[2] This assumption may be relaxed with run-time task monitoring.

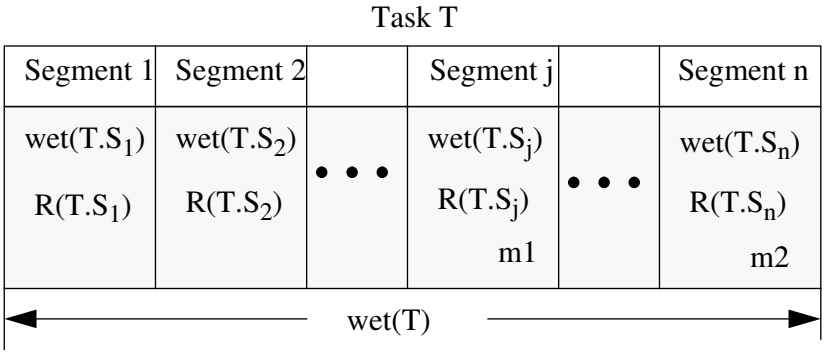

Figure 5: A Segmented Task

The *time discrepancy* of a segment is the difference between its specified execution time bound and the observed execution time; we denote the time discrepancy incurred by segment S_i of task T by $\Delta(T.S_i)$. A monitoring program is invoked when a task is activated, started or completed, a task segment is completed, and a message is produced. The actions taken by this monitoring program are as follows:

- When a message m is recognized (recognition event):
  - Queue m's characteristics into Q_F.
  - Set *mpt(m)* to *ls(m)*.
- When m's producer task T is started (start event):
  - Update *mpt(m)* by $st(T) + \sum_{i=1}^{n} wet(T.S_i)$,
    where *st(T)* stands for T's start time and
    the message m is assumed to be produced at segment S_n.
- When a segment S_k is completed (segment event):
  - Subtract $\Delta(T.S_k)$ from *mpt(m)*, where $1 \leq k < n$.
    That is, $mpt(m) = st(T) + \sum_{i=1}^{n} wet(T.S_i) - \sum_{i=1}^{k} \Delta(T.S_i)$.
- When m is produced (message production event):
  - Queue m into Q_C.
  - Remove m's characteristics from Q_F.

5.3 Assisting Task Scheduling

The execution times of real-time tasks are, in general, stochastic. Hence, estimated worst-case times are usually used for determining scheduling guarantees. Since actual execution times may be less than worst-case estimates, some processor time can be left unused. Task scheduling algorithms can be improved using information about such available processor time. Furthermore, some tasks can heavily use not only a processor but other resources as well. The set of resources used by such a task may not need to be held until task completion; for example, some resources may be used once and not again during the remaining execution. Such resources may be released early for use by other tasks. Of course, the choice of which resources can be released early may depend on run-time conditions. With information about resources that can be released early and available processor time, a task scheduler

may be able to schedule newly arrived tasks that might otherwise not be scheduled. There is a *resource reclaiming algorithm* that measures and provides underutilized yet reclaimable processor times to a task scheduler [SRS90]. However, this algorithm is invoked only when a task is completed and the next task is to be dispatched. The resources that may be released early are not considered. Integrating such an algorithm with our task segmentation and monitoring support may allow a task scheduler to schedule tasks more adaptively.

6 Performance Analysis

6.1 Simulation Model

For the evaluation of protocol performance, we introduce a simulation model, parameterized by the number of nodes and the distributions of task and message characteristics. Tasks are activated (and messages are recognized) as a Poisson process. That is, inter-task activation (and inter-message recognition) times are exponentially distributed. For simplicity, each task is assumed to produce a message when it completes. The number of segments in a task is randomly chosen from one to seven task segments. The start delay and the actual execution time of a task are uniformly distributed with mean 200 time units. Since each message has the same start delay and execution time by average, the mean of the inter-message recognition times is identical to that of the inter-message production times. We define the mean message production rate Λ to be the reciprocal of the mean inter-message production time. Message lengths are exponentially distributed with mean message length AL. A message length cannot be less than one nor greater than $10 * AL$. Message laxities are uniformly distributed in the interval $(0, laxvar * AL)$ where *laxvar* is a positive integer greater than one. We establish the estimated worst-case execution time of a task as the actual execution time plus a value randomly chosen in the interval $(lax/2, lax)$, where *lax* is the laxity of a message produced by the task. We set the latest sending time of a message to be the sum of the message laxity, the start delay, and the actual execution time of its producer task.

The system load LD is defined to be:

$$LD = \Lambda * AL * N$$

where N is the number of nodes in the system. Given LD, AL and N, the mean message production rate Λ can be obtained. The fraction of channel time used by successfully transmitted messages is known as the *effective channel utilization*. The maximum value of the effective utilization over all possible loads is known as the *capacity* of the protocol. The capacity is perhaps most greatly affected by the value of the *normalized end-to-end propagation delay* α, defined as $\alpha = \tau/AL$ [KSY84]. We measure the performance of our best-effort protocols with $\alpha = 0.1$ and 0.01 (i.e. mean message lengths are 10 and 100) and with *laxvar*= 3 and 9. The retransmission probability P of a collided message is set to be 0.5.

In real-time distributed systems, tasks in general have stochastic execution times so that the times at which they produce messages are not regular. As the execution time variance of each task increases, more future urgent messages may occur. Moreover, the asynchronous activations of the aperiodic tasks increase the irregularity of message production times. It is this irregularity that real-time protocols attempt to exploit to increase the chance of successful message transmission while decreasing the chance of message collisions. To investigate the variation in performance as the number of urgent future messages increases we introduce a parameter *SEQ*, called the *sequencer*, into our simulation model. As the sequencer value increases, the number of messages in each *cluster* tends to increase, where a

cluster is a sequence of consecutive messages produced by the same node. As a consequence, each node tends to have less chance of message collision but to have an increasing chance of having urgent future messages. Each node has an equal chance of producing n consecutive messages, where the number n is a random number drawn from the interval $(0, SEQ)$.

Each simulation experiment comprised six runs. The simulation period for each run is taken to be 5000 times the mean inter-message production time. In order to alleviate the initial bias, we collect simulation statistics after $5 * N$ messages where N is the number of nodes in the system. Each of the data points forming performance graphs shown in this section was obtained by averaging the values from these six simulation runs. Our goal is to generate a 90% confidence interval whose width is within 5% of each of these data points. Shapiro and Wilk's statistic [SW65] was used to test normality of the statistical data obtained from thirty five simulation runs; the result was insignificant at a 10% level. Applying the *t-distribution* formula [Dev87] showed that six runs were sufficient to satisfy our 5% width requirement.

6.2 Performance Measures

As the primary performance measure, we use the message loss ratio ML defined as:

$$ML = \frac{NDM}{NSM + NDM}$$

where NDM is the total number of discarded messages and NSM is the total number of successfully transmitted messages.

It can be useful to know how much performance improvement is obtained by one protocol over another in terms of message loss. We use the *performance improvement PI* for such purposes. It is defined as:

$$PI = NSM_x - NSM_y$$

where NSM_x and NSM_y respectively represent the total number of successfully transmitted messages by the protocols x and y.

The effective channel utilization ECU shows how effectively the communication channel is used. The ECU is defined as:

$$ECU = \frac{TSM}{TSIM}$$

where TSM is the total time units used for the transmission of successful messages and $TSIM$ is the total time units simulated.

6.3 Comparative Analysis

To analyze the performance of our best-effort protocols, three different classes of baseline protocols are used:

- The VTL and VTD protocols,
- The *centralized* LLF (CL) and EDF (CD) protocols. These protocols have perfect knowledge about the nodes and the channel, and can transmit messages with no collisions. That is, no time units are wasted for channel idle or collision state detection. The CL protocol shows behavior identical to that of the non-preemptive LLF task scheduler, and the CD protocol shows behavior identical to that of the non-preemptive EDF task scheduler.

- The *clairvoyant CL* and *CD* protocols. These protocols not only have perfect knowledge about the nodes and the channel but also have clairvoyancy about future messages.

While each protocol in the second class shows the best performance achievable with its own transmission policy but with no clairvoyancy, each protocol in the third class does so with clairvoyancy incorporated. Of course, the protocols in both of these classes are not realizable in practice. We label the performance curves of the clairvoyant *CL* and *CD* protocols with basic clairvoyancy *BCL* and *BCD* and those with enhanced clairvoyancy *MCL* and *MCD*. Similarly, we label the performance curves of the *BVTL* and *BVTD* protocols with basic clairvoyancy *BVTL* and *BVTD* and those with enhanced clairvoyancy *MVTL* and *MVTD*.

The performance of the virtual-time CDMA/CD protocols is shown to be not sensitive to the number of nodes but sensitive to the virtual clock rate η [ZR87]. First, we tested the *VTL* and *VTD* protocols with an increasing number of nodes; their performance is shown to be quite insensitive to the number of nodes. Then, to find the η value that gives the best *ML* and *ECU*, we tested the *VTL* and *VTD* protocols under various loads from 0.1 to 1.2 and when α=0.1 and 0.01. The best η values for both protocols are in the range of (21,27) when α=0.1 and of (31,37) when α=0.01. Based on these results, we analyze and compare the best-effort protocols with these two protocols in a system of 10 nodes. We use η=25 when $\alpha = 0.1$ and $\eta = 35$ when $\alpha = 0.01$. The value of the sequencer *SEQ* is set to be 10.

Each graph in Figure 6 shows the message loss ratio with increasing system load when α=0.1. The top two graphs (a) and (b) show the message loss ratios of the LLF-based protocols with *laxvar* 3 and 9 respectively, the bottom two graphs (c) and (d) show those of the EDF-based protocols. Corresponding to each of these graphs, the graphs in Figure 7 show the performance improvement of the best-effort protocols and the centralized protocols over the *VTL* and *VTD* protocols. While each of the BCL and BCD curves shows the maximum (ideal) performance improvement achievable with basic clairvoyancy at each given load, each of the MCL and MCD curves does so with enhanced clairvoyancy.

According to the results shown in these graphs, the best-effort protocols yield message loss ratios lower than those of the *VTL* and *VTD* protocols respectively. Only when the system has a very high load (1.0 or more), do they show similar or slightly worse performance than the *VTL* and *VTD* protocols. The overhead introduced by local and global waiting is the main cause of this phenomenon. Also, the best-effort protocols with enhanced clairvoyancy yield better performance over those with basic clairvoyancy. However, under very high or overloaded conditions (0.9 or more), the performance of the former may become poorer than that of the latter. This is because enhanced clairvoyancy results in more waiting attempts, which in turn results in increased channel overhead.

Each graph in Figure 8 and Figure 9 shows the message loss and the corresponding performance improvement in the case of $\alpha = 0.01$. With small laxities ($laxvar = 3$), the best-effort protocols always show low message loss ratios not only over the *VTL* and *VTD* protocols but also over the *CL* and *CD* protocols. Even with large laxities ($laxvar = 9$), they show lower message loss ratios until the system load is very high (0.9 or more). The performance improvement of the best-effort protocols over the *VTL* and *VTD* protocols, however, tend to degrade under high system load and overload conditions.

As mentioned earlier, as the sequencer value is increased each node tends to have fewer message collisions but to have more urgent future messages. Figure 10 shows the change in the message loss as the sequencer value is increased. While little performance change is observed in the *VTL* and *VTD* protocols, continuous performance improvements are observed in the best-effort protocols with basic and enhanced clairvoyancy. Note that each performance curve of the best-effort protocols (i.e. BVTL, BVTD, MVTL and MVTD) approaches that of the corresponding centralized ideal protocols (i.e. BCL, BCD, MCL,

and MCD) respectively as the sequencer value is increased.

Although not shown in this paper, we have also compared the effective channel utilizations of the best-effort protocols with those of the VTL and VTD protocols. When α is small (0.01), the best-effort protocols with basic and enhanced clairvoyancy (over VTL and VTD) show better utilization up to $LD = 0.5$. When the load is in the range of (0.6,1.0), depending on the message laxity and the sequencer value, utilization is sometimes better and sometimes poorer. With large α (0.1), the effective utilization of the best-effort protocols is better up to $LD = 0.4$ and sometimes better within $LD = (0.5, 0.7)$.

We have not compared the performances of the LLF-based protocols and the EDF-based protocols because the main purpose of the simulation is to compare the protocols with and without clairvoyancy. In general, the LLF-based protocols can transmit more message bits (i.e. high ECUs), while the EDF-based protocols can transmit more messages (i.e. low MLs). As indicated in [ZR87], however, the EDF-based protocols tend to be biased towards shorter length messages.

7 Conclusion

Two clairvoyant virtual-time protocols, called *best-effort* protocols, and run-time support methods allowing them to have clairvoyancy, have been described. By making use of design-time specification and run-time information about the messages and corresponding tasks, these protocols attempt to wait for and deliver urgent future messages that might otherwise be discarded.

According to simulation experiments the best-effort protocols always have better performance, in terms of message loss, than the VTL and VTD protocols under reasonable system loads (0.1 to 0.9). However, due to the time and channel overheads caused by global and local waiting, they show similar or slightly worse performance when the system load is quite high (1.0 or more). As an approach to alleviating this problem, the system could maintain system-load information which can be used by the best-effort protocols when making decisions about delivering urgent future messages. Lastly, the best-effort protocols show improving performance as the number of urgent future messages increases.

Although applied to virtual-time CSMA/CD protocols, the run-time support methods proposed here can also be applied to other communication protocols such as token passing protocols with a reservation facility. For example, in the case of token passing protocols some extra bits may need to be added in the token for message urgency indication and other necessary information.

Acknowledgments

The authors would like to thank Queen's STATLAB for their help refining the statistical methods used in this paper.

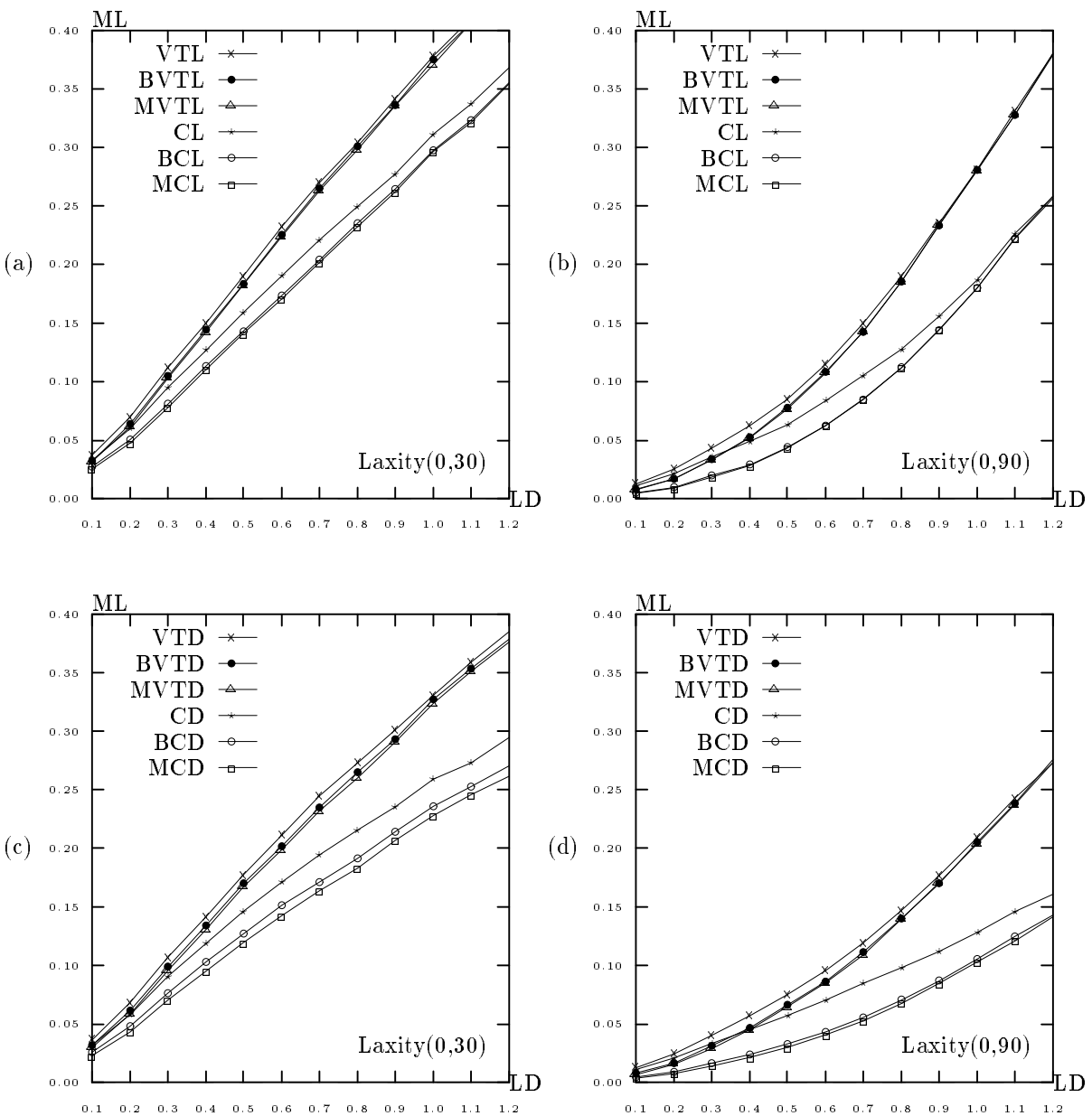

Figure 6: Message Loss ($\alpha = 0.1$)

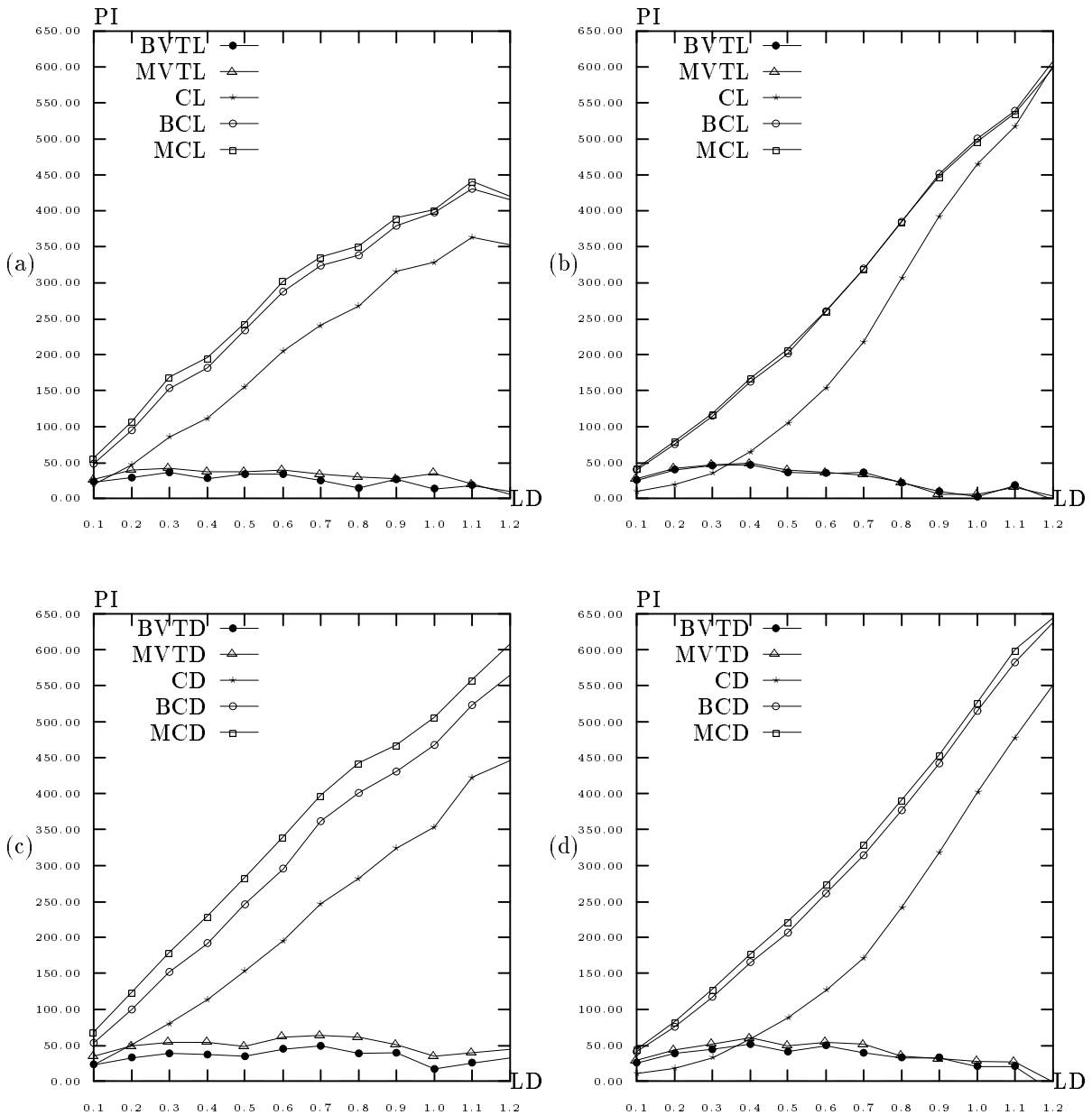

Figure 7: Performance Improvement ($\alpha = 0.1$)

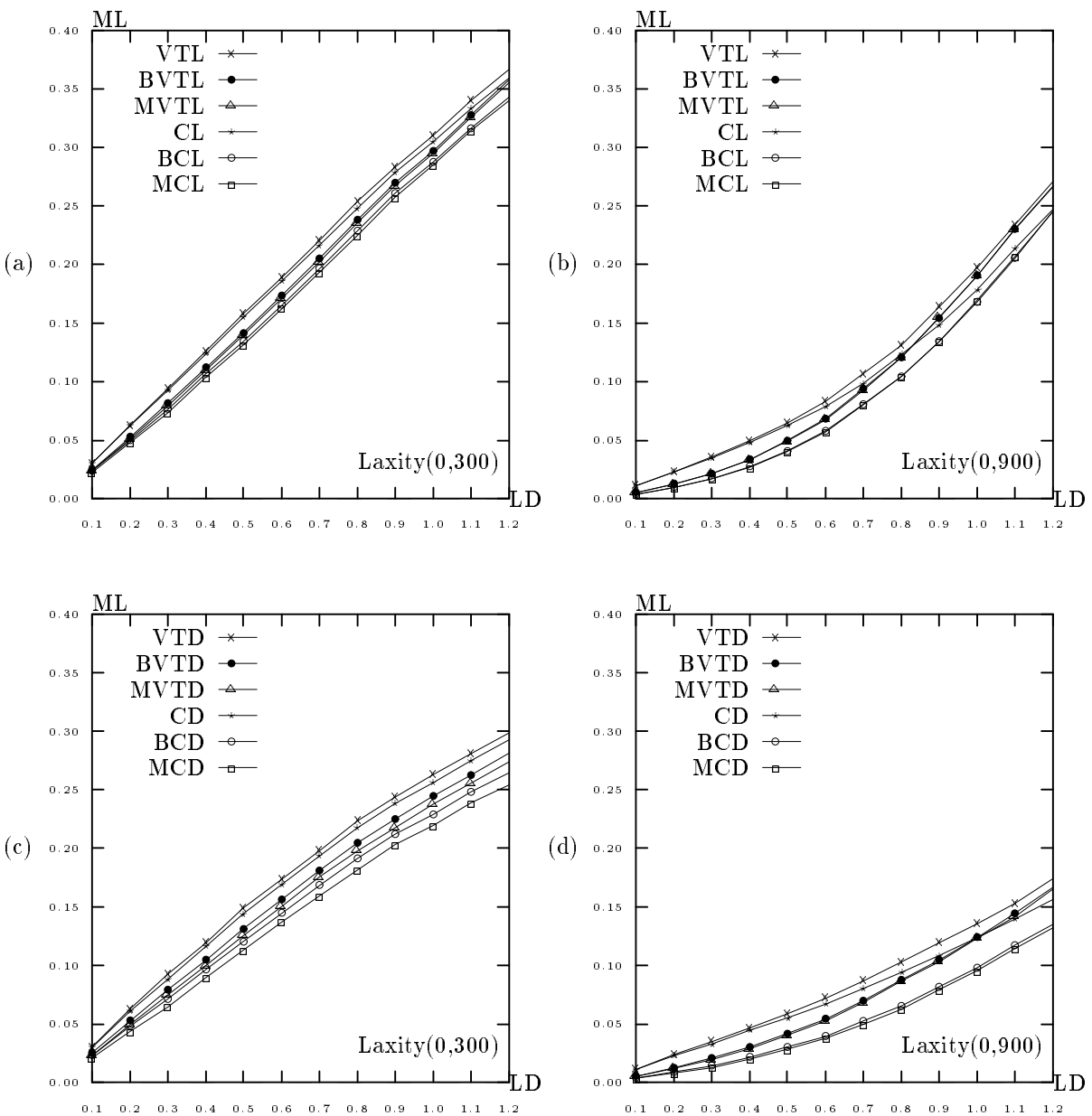

Figure 8: Message Loss ($\alpha = 0.01$)

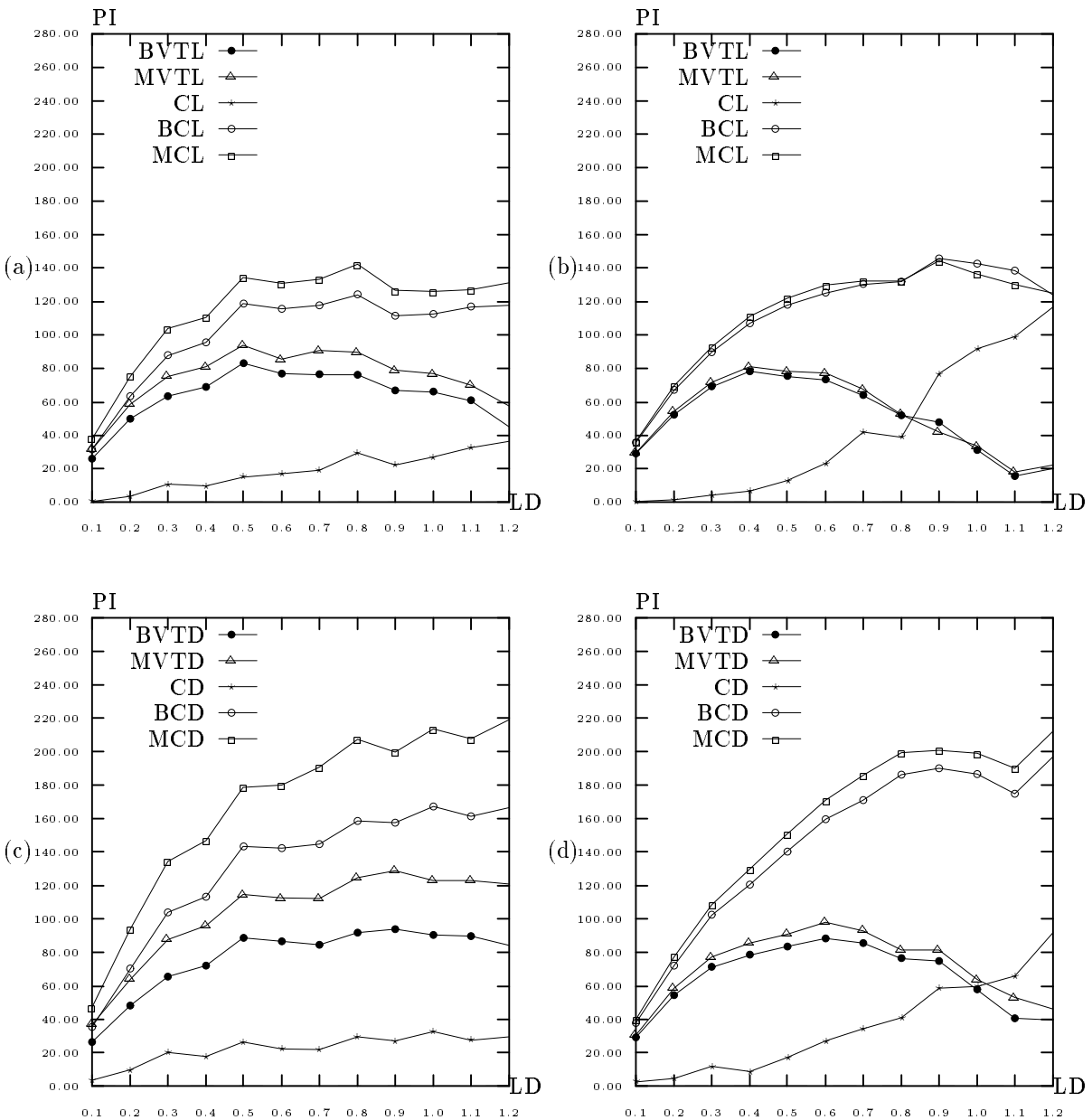

Figure 9: Performance Improvement ($\alpha = 0.01$)

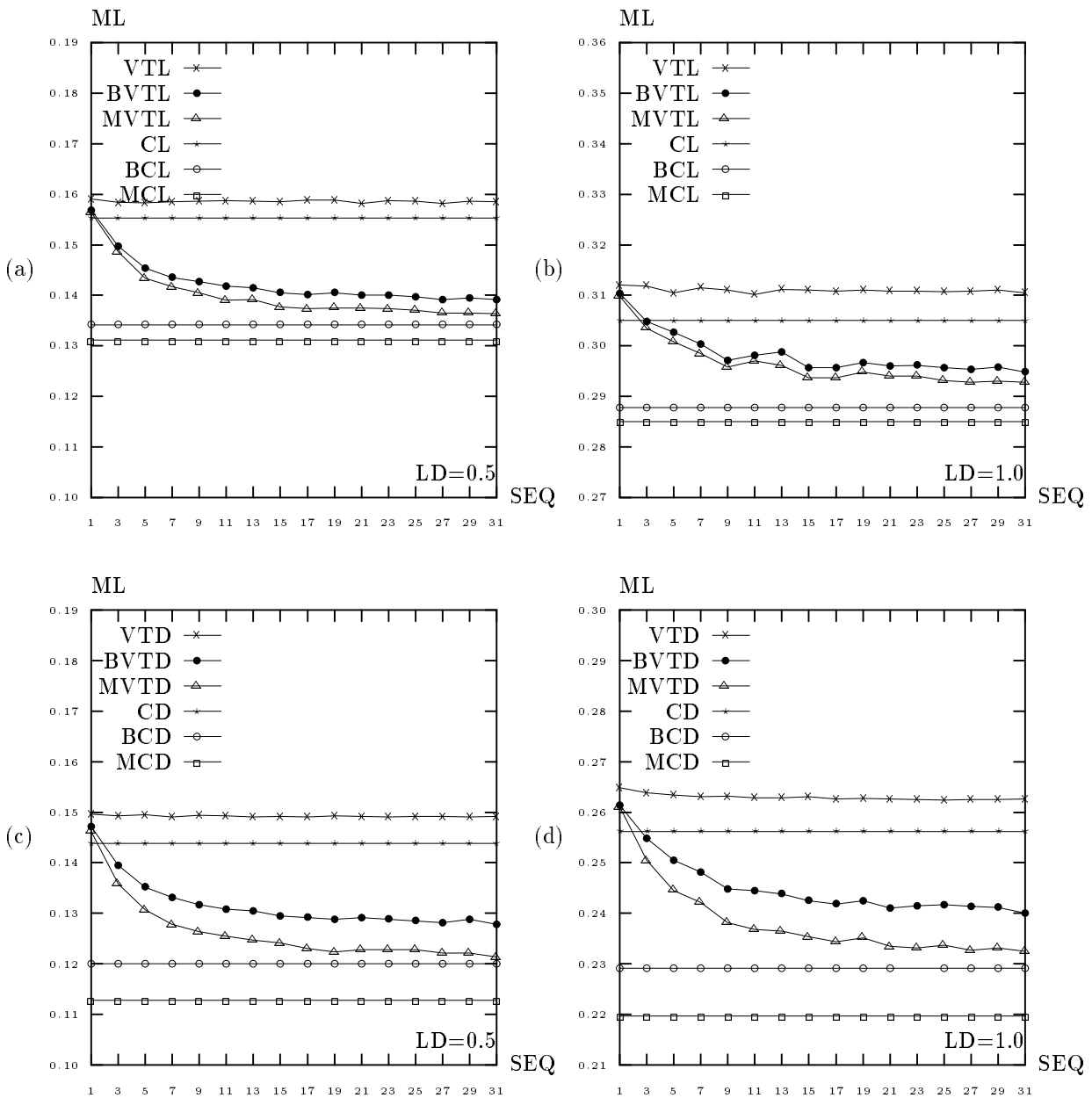

Figure 10: Message Loss with Varying Sequencer ($\alpha = 0.01$, $Laxity = (0, 300)$)

References

[Cas91] T.L. Casavant. Panel session: Debugging and performance monitoring for distributed systems: Problems and prospects. *Proceedings of the 11th International Conference on Distributed Computing Systems*, pages 378–383, May 1991.

[CJD91] S.E. Chodrow, F. Jahanian, and M. Donner. Run-time monitoring of real-time systems. *Proceedings of the Real-Time Systems Symposium*, pages 74–83, December 1991.

[Dev87] J.L. Devore. *Probability and Statistics for Engineering and the Sciences.* Brooks/Cole Publishing Company, 1987.

[DM89] M.L. Dertouzos and A.K. Mok. Multiprocessor on-line scheduling of hard real-time tasks. *IEEE Transactions on Software Engineering*, 15(12):1497–1506, December 1989. Also shown in Proceedings of the 7th Texas Conference on Computer Systems, November 1978, pages from 5-1 to 5-12.

[GG90] P. Gopinath and R. Gupta. Applying compiler techniques to scheduling in real-time systems. *Proceedings of the Real-Time Systems Symposium*, pages 247–256, December 1990.

[Gor91] M.M. Gorlick. The flight recorder: An architectural aid for system monitoring. *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 175–183, May 1991.

[HS90] D. Haban and K.G. Shin. Application of real-time monitoring to scheduling tasks with random execution times. *IEEE Transactions on Software Engineering*, 16(12):1374–1389, December 1990. Also in Real-Time Systems Symposium 1989, pages 172-181.

[HS91] W.A. Halang and A.D. Stoyenko. *Constructing Predictable Real-Time Systems.* Kluwer Academic Publishers, 1991.

[HW90] D. Haban and D. Wybranietz. A hybrid monitor for behavior and performance analysis of distributed systems. *IEEE Transactions on Software Engineering*, 16(2):197–211, February 1990.

[JG90] F. Jahanian and A. Goyal. A formalism for monitoring real-time constraints at run-time. *The 20th Symposium on Fault-Tolerant Computing*, pages 148–155, June 1990.

[JLT85] E.D. Jensen, C.D. Locke, and H. Tokuda. A time-driven scheduling model for real-time operating systems. *Proceedings of the Real-Time Systems Symposium*, pages 112–122, December 1985.

[KSY83] J.F. Kurose, M. Schwartz, and Y. Yemini. Controlling window protocols for time-constrained communication in a multiple access environment. *Proceedings of the 8th Data Communications Symposium*, 13(4):75–84, October 1983.

[KSY84] J.F. Kurose, M. Schwartz, and Y. Yemini. Multiple-access protocols and time-constrained communication. *ACM Computing Surveys*, 16(1):43–70, March 1984.

[LSMC90] J.E. Lumpp, Jr., H.J. Siegel, D.C. Marinescu, and T.L. Casavant. Specification and identification of events for debugging and performance monitoring of distributed multiprocessor systems. *Proceedings of the 10th International Conference on Distributed Computing Systems*, pages 476–483, June 1990.

[Lu82] D.J. Lu. Watchdog processors and structural integrity checking. *IEEE Transactions on Computers*, 31(7):681–685, July 1982.

[MK85] M.L. Molle and L. Kleinrock. Virtual Time CSMA: Why two clocks are better than one. *IEEE Transactions on Communications*, 33(9):919–933, September 1985.

[MM88] A. Mahmood and E.J. McClusky. Concurrent error detection using watchdog processors- a survey. *IEEE Transactions on Computers*, 37(2):160–174, February 1988.

[PTW88] S.S. Panwar, D. Towley, and J.K. Wolf. Optimal scheduling policies for a class of queues with customer deadlines to the beginning of service. *Journal of the ACM*, 35(4):832–844, October 1988.

[PZ78] L. Pouzin and H. Zimmermann. A tutorial on protocols. *Proceedings of the IEEE*, 66(11):1346–1370, November 1978.

[SR87] J.A. Stankovic and K. Ramamritham. The design of the Spring kernel. *Proceedings of the Real-Time Systems Symposium*, pages 146–157, December 1987.

[SRS90] C. Shen, K. Ramamritham, and J.A. Stankovic. Resource reclaiming in real-time. *Proceedings of the Real-Time Systems Symposium*, pages 41–50, December 1990.

[SW65] S.S. Shapiro and M.B. Wilk. An analysis of variance test for normality. *Biometrica*, 52:591–611, 1965.

[TFCB90] J.J.P. Tsai, K. Fang, H. Chen, and Y. Bi. A noninterference monitoring and replay mechanism for real-time software testing and debugging. *IEEE Transactions on Software Engineering*, 16(8):897–916, August 1990.

[TKM88] H. Tokuda, M. Kotera, and C.W. Mercer. A real-time monitor for a distributed real-time operating system. *Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, pages 68–77, May 1988.

[WS88] K. Wilken and J.P. Shen. Continuous signature monitoring: Efficient concurrent-detection of processor control-flow errors. *Proceedings on International Test Conference*, pages 914–925, September 1988.

[ZR87] W. Zhao and K. Ramamritham. Virtual time CSMA protocols for hard real-time communication. *IEEE Transactions on Software Engineering*, 13(8):938–952, 1987.

[ZSR90] W. Zhao, J.A. Stankovic, and K. Ramamritham. A window protocol for transmission of time-constrained messages. *IEEE Transactions on Computers*, 39(9):1186–1203, September 1990.