Contention-Aware Metrics: Analysis of Distributed Algorithms

Péter Urbán* Xavier Défago André Schiper
peter.urban@epfl.ch *xavier.defago@epfl.ch* *andre.schiper@epfl.ch*

Département de Systèmes de Communication
École Polytechnique Fédérale de Lausanne
1015 Lausanne EPFL, Switzerland
fax: +41-21-693 6770
`http://lsewww.epfl.ch/`

Abstract

Resource contention is widely recognized as having a major impact on the performance of distributed algorithms. Nevertheless, the metrics that are commonly used to predict their performance take little or no account of contention. In this paper, we define two performance metrics for distributed algorithms that account for network contention as well as CPU contention. We then illustrate the use of these metrics by comparing four Atomic Broadcast algorithms, and show that our metrics allow for a deeper understanding of performance issues than conventional metrics.

1 Introduction

Performance prediction and evaluation are a central part of every scientific and engineering activity including the construction of distributed applications. Engineers of distributed systems rely heavily on various performance evaluation techniques and have developed the necessary techniques for this activity. In contrast, algorithm designers invest considerable effort in proving the correctness of their algorithms (which they of course should do!), but often oversee the importance of predicting the performance of their algorithms, i.e., they rely on simplistic metrics. As a result, there is a serious gap between the prediction and the evaluation of performance of distributed algorithms.

Performance Prediction vs. Evaluation of Algorithms. When analyzing performance, one has to make a distinction between prediction and evaluation. Performance prediction gives an indication of the expected performance of an algorithm, *before* it is actually implemented. Performance prediction techniques give fairly general yet imprecise information, and rely on the use of various metrics. Conversely, performance evaluation is an *a posteriori* analysis of an algorithm, once it has

*Péter Urbán is the contact author.

been implemented and run in a given environment (possibly a simulation). While the information obtained is usually very accurate and precise, the results depend on specific characteristics of the environment and thus lack generality. Performance prediction and evaluation are complementary techniques. Performance prediction is used to orient design decisions, while performance evaluation can confirm those decisions and allows the dimensioning of the various system parameters.

Definition of a Metric. In this paper, we focus on the problem of predicting and comparing the performance of distributed algorithms. The goal is hence to investigate metrics that answer typical questions such as choosing the best algorithm for a particular problem, or identifying the various performance tradeoffs related to the problem. We define a *metric* as a value associated with an algorithm, that has no physical reality and is used to define an order relation between algorithms. A good metric should provide a good approximation of the performance of an algorithm, regardless of the implementation environment. Even though some performance evaluation techniques are also based on an abstract model of the system (e.g., analytical approaches, simulation), a metric must be a *computable* value. This is clearly in contrast with simulation techniques that can model details of the system and the environment, and thus use complex models.

Existing Metrics for Distributed Algorithms. As mentioned earlier, performance prediction of distributed algorithms is usually based on two rather simplistic metrics: time and message complexity. These metrics are indeed useful, but there is still a large gap between the accuracy of the information they provide, and results obtained with more environment specific approaches.

The first commonly used metric, called *time complexity*, measures the latency of an algorithm. There exist many definitions of time complexity that are more or less equivalent. A common way to measure the time complexity of an algorithm (e.g., [1, 24, 21, 17, 13, 23]) consists in considering the algorithm in a model where the message delay has a known upper bound δ. The efficiency of the algorithm is measured as the maximum time needed by the algorithm to terminate. This efficiency is expressed as a function of δ, and is sometimes called the latency degree. This metric is *latency-oriented*, i.e., measures the cost of *one* execution of the algorithm.

The second metric, called *message complexity*, consists in counting the total number of messages generated by the algorithm [21, 13, 1]. This metric is useful when combined with time complexity, since two algorithms that have the same time complexity can generate a different volume of messages. Knowing the number of messages generated by an algorithm gives a good indication of its scalability and the amount of resources it uses. Furthermore, an algorithm that generates a large number of messages is likely to generate a high level of network contention.

Resource Contention. Resource contention is often a limiting factor for the performance of distributed algorithms. In a distributed system, the key resources are (1) the CPUs and (2) the network, any of which is a potential bottleneck. The major weakness of the time and message complexity metrics is that neither attaches enough importance to the problem of resource contention. While the message complexity metric ignores the contention on the CPUs, the time complexity metric ignores contention completely.

Contribution and Structure. In this paper, we define two metrics (one latency-oriented, the other throughput-oriented) which account for resource contention, both on the CPUs and the network. The use of those metrics is then illustrated by comparing Atomic Broadcast algorithms. The rest of the paper is structured as follows. Section 2 presents related work. In Section 3, we present the system model on which our metrics are based. Section 4 presents a *latency-oriented* and a *throughput-oriented* metric. We then compare some algorithms using our metrics in Section 5, and discuss the results. Because of space constraints, the detailed analysis appears in the Appendix. Finally, Section 6 concludes the paper.

2 Related Work

Resource Contention in Network Models. Resource contention (also sometimes called congestion) has been extensively studied in the literature. The bulk of the publications about resource contention describe strategies to either avoid or reduce resource contention (e.g. [14, 15]). Some of this work analyze the performance of the proposed strategies. However, these analyses consist in performance *evaluation*, and use models that are often specific to a particular network (e.g., [20]). Distributed algorithms are normally developed assuming the availability of some transport protocol. A metric that compares these algorithms must abstract out details that are only relevant to some implementations of a transport layer. In other words, it is necessary to relinquish precision for the sake of generality.

Resource Contention in Parallel Systems. Dwork, Herlihy, and Waarts [11] propose a complexity model for shared-memory multiprocessors that takes contention into account. This model is very interesting in the context of shared memory systems but is not well suited to the message passing model that we consider here. The main problem is that the shared memory model is a high-level abstraction for communication between processes. As such, it hides many aspects of communication that are important in distributed systems. Dwork, Herlihy, and Waarts associate a unit cost based on the access to shared variables, which has a granularity too coarse for our problem.

Computation Models for Parallel Algorithms. Unlike distributed algorithms, many efforts have been directed at developing performance prediction tools for parallel algorithms. However, the execution models are not adapted to distributed algorithms: for instance, the PRAM model (e.g., [18]) requires that processors evolve in locksteps and communicate using a global shared memory; the BSP model [28] requires that processors communicate using some global synchronization operation; the LogP model [9] assumes that there is an absolute upper bound on the transmission delay of messages. Hence, these models are not adequate to predict the perofrmance of distributed algorithms. For instance, metrics developed in these models are too coarse grained for many asynchronous distributed algorithms, such as group communication protocols.

Competitive Analysis. Other work, based on the method of competitive analysis proposed by Sleator and Tarjan [26], has focused on evaluating the competitiveness of distributed algorithms [5, 6]. In this work, the cost of a distributed algorithm is compared to the cost of an optimal centralized algorithm with a global knowledge. This work has been refined in [2, 3, 4] by considering an optimal *distributed* algorithm as the reference for the comparison. This work assumes an asynchronous shared-memory model and predicts the performance of an algorithm by counting the number of steps required by the algorithms to terminate. The idea of evaluating distributed algorithms against an optimal reference is appealing, but this approach is orthogonal to the definition of a metric. The metric used is designed for the shared-memory model, and still ignores the problem of contention.

3 Distributed System Model

The two metrics that we define in this paper are based on an abstract system model which introduces two levels of resource contention: *CPU contention* and *network contention*. First, we define a basic version of the model that leaves some aspects unspecified, but is sufficient to define our throughput oriented metric (see Definition 5). Second, we define an extended version of the model by lifting the ambiguities left in the basic version. This extended model is used in Sect. 4 to define our latency oriented metric (see Definition 3).

3.1 Basic Model

The model is inspired from the models proposed in [25, 27]. It is built around of two types of resources: CPU and network. These resources are involved in the transmission of messages between processes. There is only one network that is shared among processes, and is used to transmit a message from one process to another. Additionally, there is one CPU resource attached to each process in the system. These CPU resources represent the processing performed by the network controllers and the communication layers, during the emission and the reception of a message. In this model, the cost of running the distributed algorithm is neglected, and hence this does not require any CPU resource.

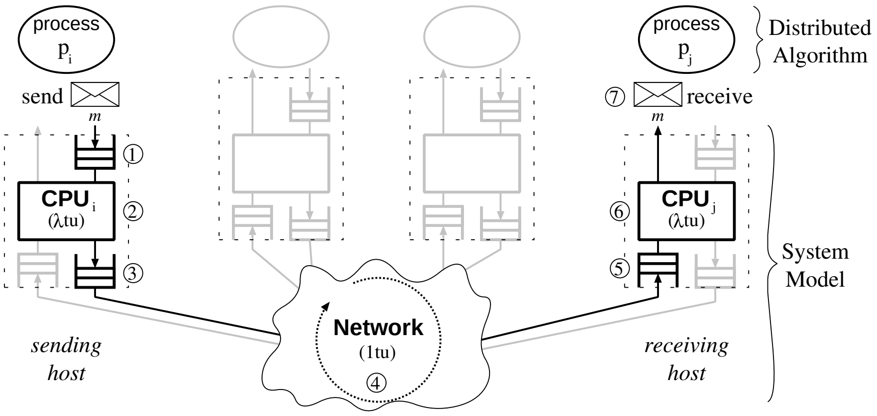


Figure 1: Decomposition of the end-to-end delay (tu=time unit).

The transmission of a message m from a sending process p_i to a destination process p_j occurs as follows (see Fig. 1):

1. m enters the *sending queue*[1] of the sending host, waiting for CPU_i to be available.

[1]All queues in the model use a FIFO policy (sending, receiving, and network queues).

2. m takes the resource CPU_i for λ time units, where λ is a parameter of the system model ($\lambda \geq 0$).

3. m enters the *network queue* of the sending host and waits until the network is available for transmission.

4. m takes the network resource for 1 time unit.

5. m enters the *receive queue* of the destination host and waits until CPU_j is available.

6. m takes the resource CPU_j of the destination host for λ time units.

7. Message m is received by p_j in the algorithm.

3.2 Extended Model

The basic model is not completely specified. For instance, it leaves unspecified the way some resource conflicts are resolved. We now extend the definition of the model so that to specify these points. As a result, the execution of a (deterministic) distributed algorithm in the extended system model is *deterministic*.

Network. Concurrent requests to the network may arise when messages at different hosts are simultaneously ready for transmission. The access to the network is modeled by the following code (executed by the network):

```
i ← 1
loop
    wait until one network queue is not empty
    while network queue of CPU_i is empty do
        increment i (mod n)
    end
    m ← extract 1^st msg from network queue of CPU_i
    wait 1 time unit
    insert m into receiving queue of CPU_dest(m)
    increment i (mod n)
end loop
```

CPU. CPU resources also appear as points of contention between a message in the sending queue and a message in the receiving queue. This issue is solved by giving priority on every host to outgoing messages over incoming ones.

Send to all. Distributed algorithms often require to send a message m to all processes, using a "send to all" primitive. The way this is actually performed depends on the model (see below).

Definition 1 (point-to-point) *Model $\mathcal{M}(n, \lambda)$ is the extended model with parameters n and λ, where $n > 0$ is the number of processes and $\lambda \geq 0$ is the relative cost between CPU and network. The primitive "send to all" is defined as follows: If p is a process that sends a message m to all processes, then p sends the message m consecutively to all processes in the lexicographical order ($p_1, p_2, \ldots, p_n$).*

Nowadays, many networks are capable of broadcasting information in an efficient manner, for instance, by providing support for IP multicast [10]. For this reason, we also define a model that integrates the notion of a broadcast network.

Definition 2 (broadcast) *Model $\mathcal{M}_{br}(n, \lambda)$ is defined similarly to Definition 1, with the exception of the "send to all" primitive, which is defined as follows: If p is a process that sends a message m to all, then p sends a single copy of m, the network transmits a single copy of m, and each process (except p) receives a copy of m.*

3.3 Illustration

Let us now illustrate the model with an example. We consider a system with three processes $\{p_1, p_2, p_3\}$ which execute the following simple algorithm. Process p_1 starts the algorithm by sending a message m_1 to processes p_2 and p_3. Upon reception of m_1, p_2 sends a message m_2 to p_1 and p_3, and p_3 sends a message m_3 to p_1 and p_2.

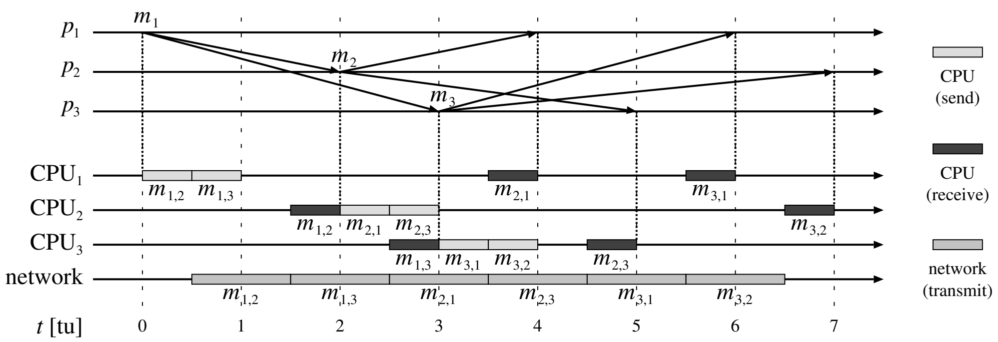


Figure 2: Simple algorithm in model $\mathcal{M}(3, 0.5)$ ($m_{i,j}$ denotes the copy of message m_i sent to process p_j).

Figure 2 shows the execution of this simple algorithm in model $\mathcal{M}(3, 0.5)$. The upper part of the figure is a time-space diagram showing the exchange of messages between the three processes. The lower part is a more detailed diagram that shows the activity (send, receive, transmit) of each resource in the model. For instance, process p_3 sends a copy of message m_3 to process p_1 (denoted $m_{3,1}$) at time 3. The message takes the CPU resource of p_3 at time 3, takes the network resource at time 4.5, and takes the CPU resource of p_1 at time 5.5. Finally, m_3 is received by p_1 at time 6.

4 Contention-Aware Metrics

4.1 Latency Metric

The definition of the latency metric uses the terms: "start" and "end" of a distributed algorithm. These terms are supposed to be defined by the problem $\mathcal{P}$ that an algorithm $\mathcal{A}$ solves. They are not defined as a part of the metric.

Definition 3 (latency metric) *Let $\mathcal{A}$ be a distributed algorithm. The latency metric* $\mathrm{Latency}(\mathcal{A})(n, \lambda)$ *is defined as the number of time units that separate the start and the end of algorithm $\mathcal{A}$ in model $\mathcal{M}(n, \lambda)$.*

Definition 4 (latency metric (broadcast)) *Let $\mathcal{A}$ be a distributed algorithm. The definition of the latency metric* $\mathrm{Latency}_{br}(\mathcal{A})(n, \lambda)$ *is the same than Definition 3, but in model $\mathcal{M}_{br}(n, \lambda)$.*

4.2 Throughput Metric

The throughput metric of an algorithm $\mathcal{A}$ considers the utilization of system resources in one run of $\mathcal{A}$. The most heavily used resource constitutes a bottleneck, which puts a limit on the *maximal throughput*, defined as an upper bound on the frequency at which the algorithm can be run.

Definition 5 (throughput metric) *Let $\mathcal{A}$ be a distributed algorithm. The throughput metric is defined as follows:*

$$\mathrm{Thput}(\mathcal{A})(n, \lambda) \stackrel{\mathrm{def}}{=} \frac{1}{\max_{r \in \mathcal{R}_n} T_r(n, \lambda)}$$

where $\mathcal{R}_n$ denotes the set of all resources (i.e., $\mathrm{CPU}_1, \ldots, \mathrm{CPU}_n$ *and the network), and $T_r(n, \lambda)$ denotes the total duration for which resource $r \in \mathcal{R}_n$ is utilized in one run of algorithm $\mathcal{A}$ in model $\mathcal{M}(n, \lambda)$.*

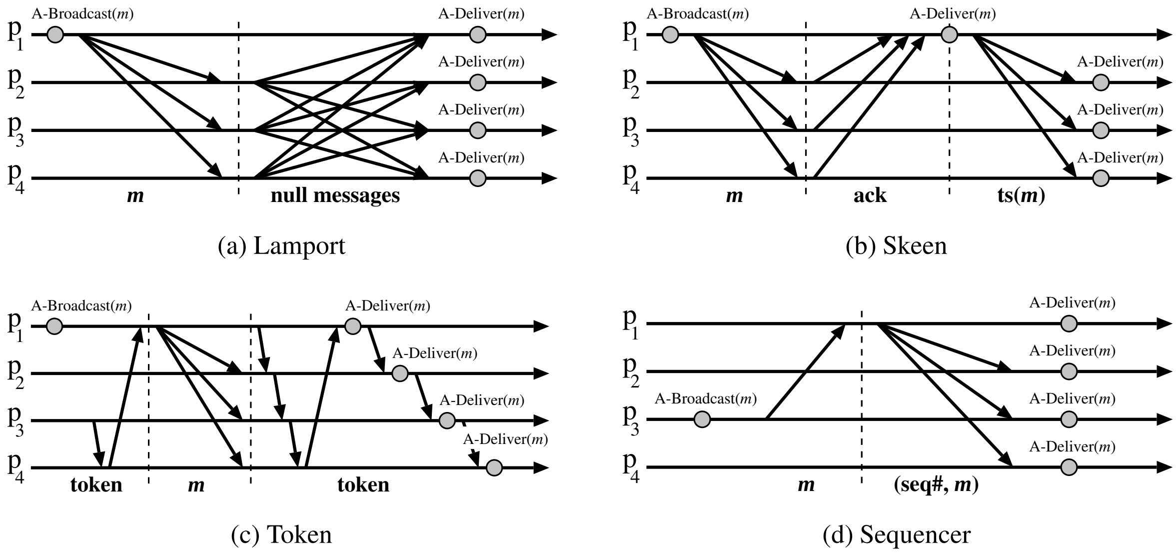
$\mathrm{Thput}(\mathcal{A})(n, \lambda)$ can be understood as an upper bound on the frequency at which algorithm $\mathcal{A}$ can be started. Let r_b be the resource with the highest utilization time: $T_{r_b} = \max_{r \in \mathcal{R}_n} T_r$. At the frequency given by $\mathrm{Thput}(\mathcal{A})(n, \lambda)$, r_b is utilized at 100%, i.e., it becomes a bottleneck.

Definition 6 (throughput metric (broadcast)) *Let $\mathcal{A}$ be a distributed algorithm. The definition of the throughput metric* $\mathrm{Thput}_{br}(\mathcal{A})(n, \lambda)$ *is the same than Definition 5, but in model $\mathcal{M}_{br}(n, \lambda)$.*

Relation with Message Complexity. The throughput metric can be seen as a generalization of message complexity. While our metric considers different types of resources, message complexity only considers the network. It is easy to see that $T_{network}$, the utilization time of the network in a single run, gives the number of messages exchanged in the algorithm.

5 Comparison of Atomic Broadcast Algorithms

We now illustrate the use of our two metrics by comparing four different algorithms that solve the problem of *Atomic Broadcast*. These examples show that our metrics yield results that are more precise than what can be obtained by relying solely on time and message complexity. This clearly confirms the observation that contention is a factor that cannot be overlooked.



(a) Lamport

(b) Skeen

(c) Token

(d) Sequencer

Figure 3: Communication patterns of Atomic Broadcast algorithms.

5.1 Atomic Broadcast Algorithms

Atomic Broadcast is fundamental problem in the context of distributed systems [13]. Informally, the problem consists in broadcasting messages to other processes, in such a way that all messages are delivered in the same order by all destination processes. The problem is defined in terms of the two events *A-Broadcast* and *A-Deliver*. When a process wants to atomically broadcast a message m it executes A-Broadcast(m), and A-Deliver(m) executed by process q corresponds to the delivery of message m by q. The latency of the algorithm with respect to message m is then defined as follows. We consider a run in which no other message is A-Broadcast; the algorithm starts when a process executes A-Broadcast(m) and ends when the last process executes A-Deliver(m).

We briefly describe four different Atomic Broadcast algorithms for a system with no failures, and compare them using our metrics. Figure 3 shows the communication pattern associated with the broadcast of a single message m for each of the four algorithms. Note that the communication pattern is enough to compute our metrics. For this reason, we have omitted to give the details of each algorithm.

Lamport. In Lamport [19], every message carries a logical time-stamp (see [19]). To atomically broadcast a message m, the sender process first sends m to all other processes (see Fig. 3(a)). Upon reception of m, a process p sends a time-stamped "null message" to all others, thus informing them that it has no other message that may have to be delivered before m. These null messages appear only when a process has nothing to broadcast.

Skeen. Skeen's algorithm (described in [8, 12]) is a two-phase protocol that can use Lamport's logical clocks [19]. To atomically broadcast a message m, a process p first sends m to all processes (see Fig. 3(b)). Upon reception of m, the processes send a time-stamped acknowledgment message to p. Once p has received all acknowledgments, it takes the maximum of the timestamps received, and sends this information to all processes. Processes deliver m after they receive this message (the details of the delivery condition is irrelevant here).

Token. In Rajagopalan and McKinley's token-based algorithm [22], a token circulates in the system and a process is allowed to broadcast messages only when it holds the token. To atomically broadcast a message m, a process p must first wait for the token[2] (see Fig. 3(c)). When it holds the token, p broadcasts m to the other processes and passes the token to the next process. The message m can be delivered only after it has been acknowledged by all processes. The acknowledgments of messages are carried by the token. So m is delivered by the last process only after two round-trips of the token.

Sequencer. Many Atomic Broadcast algorithms are based on the principle that one process is designated as a sequencer and constructs the order (e.g., [7, 16]). In the version that we consider here (see Fig. 3(d)), a process atomically broadcasts a message m by sending m to the sequencer. Upon reception of m, the sequencer attaches a sequence number to m and sends it to all other processes. Messages are then delivered according to their sequence number.

These algorithms are interesting to illustrate our metrics because they take contrasting approaches to solve the problem of Atomic Broadcast. Although they all deliver messages according to some total order, these algorithms actually provide varying levels of guarantees, and are hence not equivalent. An actual comparison must of course take these issues into account.

5.2 Latency Metric

We now analyze the latency of the four Atomic Broadcast algorithms: Lamport, Skeen, Token, and Sequencer. For each algorithm, we compute the value of the latency metric in model $\mathcal{M}(n, \lambda)$. The results are summarized in Table 1 and compared in Fig. 4(a).[3] Table 1 also shows the time complexity of the algorithms. For time complexity, we use the *latency degree* [24]: roughly speaking, an algorithm with latency degree l requires l communication steps.

Table 1: Latency metric: evaluation of Atomic Broadcast algorithms (in model $\mathcal{M}(n, \lambda)$)

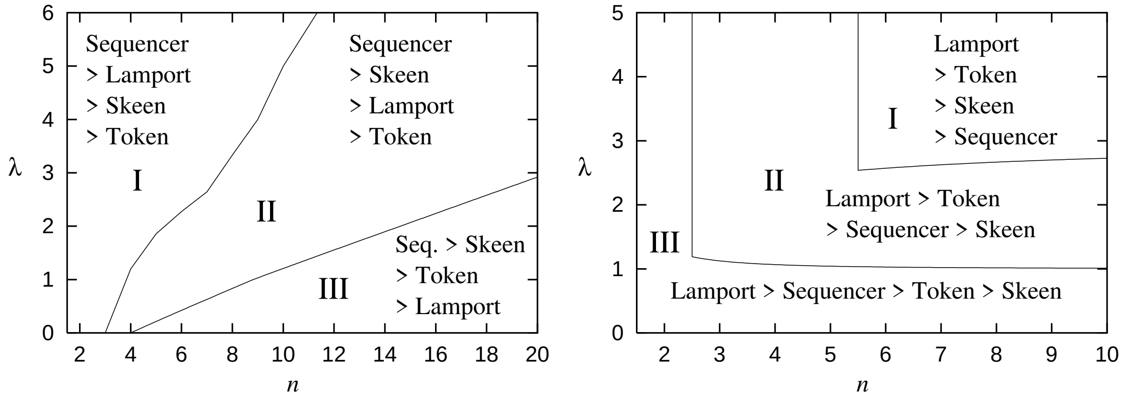
Algorithm $\mathcal{A}$	$\mathrm{Latency}\,(\mathcal{A})(n, \lambda)$		**Time complexity**
Lamport	$\approx 3(n-1)\lambda + 1$	if $n \leq \lambda + 2$	2
	$\approx \frac{1}{2}n(n-3) + 2\lambda n + \frac{1}{2}\lambda^2 - \frac{3}{2}\lambda$	if $n \leq 2\lambda + 3$	
	$\approx \frac{1}{2}n(n-1) + 2\lambda n + \lambda^2 - \frac{7}{2}\lambda - 3$	if $n \leq 4\lambda - 4$	
	$\approx n(n-1) + \lambda^2 + \lambda + 5$	otherwise	
Skeen	$\approx 3(n-1) + 4\lambda$	if $\lambda < 1$	3
	$\approx (3n-2)\lambda + 1$	if $\lambda \geq 1$	
Token	$(2.5n-1)(2\lambda+1) + \max(1,\lambda)(n-1)$		$2.5n - 1$
Sequencer	$4\lambda + 2 + \max(1,\lambda)(n-2)$		2

Figure 4(a) represents the results of the comparison between the four algorithms with respect to the latency metric. The area is split into three zones in which algorithms perform differently with respect to each other (e.g., in Zone I, we have Sequencer $>$ Lamport $>$ Skeen $>$ Token, where $>$

[2] In our analysis, we take the average case where the token is always halfway on its path toward p.

[3] For reasons of clarity, we give approximate values for $\mathrm{Latency}\,(\mathrm{Lamport})(n, \lambda)$ and $\mathrm{Latency}\,(\mathrm{Skeen})(n, \lambda)$. The expressions given for these two algorithms ignore a factor that is negligible compared to the rest of the expression. The exact expressions, as well as a description of the analysis are given in Appendix A.

means "better than"). The latency metric and time complexity yield the same results for three of the four algorithms: Token, Skeen, and Sequencer. Both metrics yield that Sequencer performs better than Skeen, which in turn performs better than Token. For Lamport, time complexity suggests that it always performs better than the other algorithms. This comes in contrast with our latency metric which shows that the relative performance of Lamport are clearly dependent on the system parameters n and λ. The reason is that Lamport generates a quadratic number of messages and is hence subject to network contention to a greater extent. Time complexity is unable to predict this as it fails to account for contention.



(a) $\mathrm{Latency}(\mathcal{A})(n, \lambda)$ (b) $\mathrm{Thput}(\mathcal{A})(n, \lambda)$

Figure 4: Comparison of Atomic Broadcast algorithms ($\mathcal{A} > \mathcal{A}'$ means $\mathcal{A}$ "better than" $\mathcal{A}'$).

5.3 Throughput Metric

We now analyze the throughput of the four algorithms. In a throughput analysis, one run of the algorithm should not be considered in isolation. Indeed, many algorithms behave differently whether they are under high load or not (e.g., Lamport does not need to generate null messages under high load). For this reason, the throughput metric is computed by considering a run of the algorithm *under high load*. We also assume that every process atomically broadcasts messages, and that the emission is fairly distributed among them. For each algorithm, we compute the value of the throughput metric in model $\mathcal{M}(n, \lambda)$. The results are summarized[4] in Table 2. The algorithms are then compared in Fig. 4(b).

Figure 4(b) illustrates the relative throughput of the four algorithms. The graph is split into three zones in which algorithms perform differently with respect to each other. The throughput metric and message complexity both yield that Lamport performs better than Token which in turn performs better than Skeen. However, the two metrics diverge when considering Sequencer. Indeed, while message complexity suggest that Sequencer always performs better than Skeen and Token, our throughput metric clearly shows that it is not always the case. In fact, Sequencer is

[4] The full description of the analysis is given in Appendix B.

Table 2: Throughput metric: evaluation of Atomic Broadcast algorithms (in model $\mathcal{M}(n, \lambda)$)

Algorithm $\mathcal{A}$	$(\mathrm{Thput}(\mathcal{A})(n, \lambda))^{-1}$	**Message complexity**
Lamport	$(n-1) \cdot \max\left(1, \frac{2\lambda}{n}\right)$	$n-1$
Skeen	$3(n-1) \cdot \max\left(1, \frac{2\lambda}{n}\right)$	$3(n-1)$
Token	$n \cdot \max\left(1, \frac{2\lambda}{n}\right)$	n
Sequencer	$\left(n - \frac{1}{n}\right) \cdot \max(1, \lambda)$	$n - \frac{1}{n}$

more subject to CPU contention than the other three algorithms. This type of contention is especially noticeable in systems with large values of λ. Message complexity fails to pinpoint this, as it does not take CPU contention into account.

5.4 Latency and Throughput in Broadcast Networks

The analysis in model $\mathcal{M}_{br}(n, \lambda)$ are not much different. In fact, there are less messages and

Table 3: $\mathrm{Latency}_{br}(\mathcal{A})(n, \lambda)$: evaluation of Atomic Broadcast algorithms.

Algorithm $\mathcal{A}$	$\mathrm{Latency}_{br}(\mathcal{A})(n, \lambda)$
Lamport	$4\lambda + n$
Skeen	$6\lambda + 3 + (n-2) \cdot \max(1, \lambda)$
Token	$\left(\frac{5n}{2} - 1\right)(2\lambda + 1) + \max(1, \lambda)$
Sequencer	$4\lambda + 2$

Table 4: $\mathrm{Thput}_{br}(\mathcal{A})(n, \lambda)$: evaluation of Atomic Broadcast algorithms.

Algorithm $\mathcal{A}$	$(\mathrm{Thput}_{br}(\mathcal{A})(n, \lambda))^{-1}$	**Msg complexity**
Lamport	$\max(1, \lambda)$	1
Skeen	$\max\left(n+1, \frac{4n+1}{n}\lambda\right)$	$n+1$
Token	$\max\left(2, \frac{n+2}{n}\lambda\right)$	2
Sequencer	$\frac{2n-1}{n} \max(1, \lambda)$	$2 - \frac{1}{n}$

less contention, and thus yields results that are a little less spectacular.[5] Table 3 and Table 4 show the results of the two metrics in a broadcast network ($\mathrm{Latency}_{br}(\mathcal{A})(n, \lambda)$ and $\mathrm{Thput}_{br}(\mathcal{A})(n, \lambda)$). Apart from the fact that these results are simpler than in a model with point-to-point communication, there are interesting differences.

According to the latency metric, for any "realistic" value[6] of λ and n, the algorithms are always ordered as follows:

$$\mathrm{Sequencer} > \mathrm{Lamport} > \mathrm{Skeen} > \mathrm{Token}$$

Unlike the results obtained with $\mathrm{Latency}(\mathcal{A})(n, \lambda)$, there is only one single zone with a broadcast network. This zone corresponds to zone I depicted on Figure 4(a) but, in model $\mathcal{M}_{br}(n, \lambda)$, the algorithms are not ordered differently as n increases. This is easily explained by the fact that Lamport is quadratic in model $\mathcal{M}(n, \lambda)$ while it is linear in model $\mathcal{M}_{br}(n, \lambda)$. The latency of the three other algorithms is not so different because they are linear in both models.

[5] The full description of the analysis in model $\mathcal{M}_{br}(n, \lambda)$ is given in Appendix C.

[6] Realistic values for the parameters λ and n are: $\lambda \geq 0$ and $n \geq 2$.

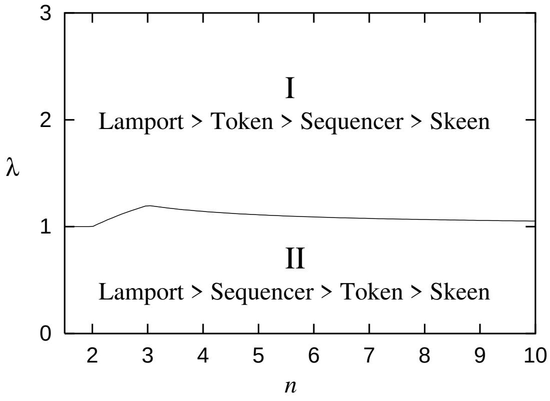


Figure 5: Comparison of the throughput of Atomic Broadcast algorithms in a broadcast network (using $\mathrm{Thput}_{br}(\mathcal{A})(n, \lambda)$). ($\mathcal{A} > \mathcal{A}'$ means $\mathcal{A}$ "better than" $\mathcal{A}'$).

Similarly, $\mathrm{Thput}_{br}(\mathcal{A})(n, \lambda)$ yields simpler results than $\mathrm{Thput}(\mathcal{A})(n, \lambda)$. As shown in Figure 5, the parameter space is cut into two zones (instead of three for $\mathrm{Thput}(\mathcal{A})(n, \lambda)$, as shown on Fig. 4(b)). The difference between the two zones is the relative performance (throughput) of Sequencer and Token. This yields that Token is better than Sequencer when the CPU is a limiting factor. In fact, Sequencer is limited by the sequencer process which becomes a clear bottleneck. Conversely, Token spreads the load evenly among all processes, and so none becomes a bottleneck. Once again, both classical metrics (time and message complexity) fail to capture this aspect.

6 Conclusion

The paper proposes two metrics to predict the latency and the throughput of distributed algorithms. Unlike other existing metrics, the two complementary metrics that we present here take account of both network and CPU contention. This allows for more precise predictions and a finer grained analysis of algorithms than what time complexity and message complexity permit. In addition, our metrics make it possible to find out whether the bottleneck is the network or the CPU of one specific process.

The problem of resource contention is commonly recognized as having a major impact on the performance of distributed algorithms. Because other metrics do not take account of contention to the same extent as ours, our metrics fill a gap that exists between simple complexity measures and more complex performance evaluation techniques.

Acknowledgments

We would like to thank Prof. Jean-Yves Le Boudec for his numerous comments and advice on early versions of this work.

References

[1] M. K. Aguilera, W. Chen, and S. Toueg. Failure detection and consensus in the crash-recovery model. In *Proc. 12th Int'l Symp. on Distributed Computing (DISC)*, pages 231–245, Sept. 1998.

[2] M. Ajtai, J. Aspnes, C. Dwork, and O. Waarts. A theory of competitive analysis for distributed algorithms. In S. Goldwasser, editor, *Proc. 35th Annual Symp. on Foundations of Computer Science*, pages 401–411, Nov. 1994.

[3] J. Aspnes and O. Waarts. A modular measure of competitiveness for distributed algorithms (abstract). In *Proc. 14th ACM Symp. on Principles of Distributed Computing (PODC)*, page 252, Aug. 1995.

[4] J. Aspnes and O. Waarts. Modular competitiveness for distributed algorithms. In *Proc. 28th ACM Symp. on Theory of Computing (STOC)*, pages 237–246, May 1996.

[5] B. Awerbuch, S. Kutten, and D. Peleg. Competitive distributed job scheduling. In *Proc. 24th ACM Symp. on Theory of Computing (STOC)*, pages 571–580, May 1992.

[6] Y. Bartal, A. Fiat, and Y. Rabani. Competitive algorithms for distributed data management. In *Proc. 24th ACM Symp. on Theory of Computing (STOC)*, pages 39–50, May 1992.

[7] K. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Trans. Comput. Syst.*, 9(3):272–314, Aug. 1991.

[8] K. P. Birman and T. A. Joseph. Reliable communication in presence of failures. *ACM Trans. Comput. Syst.*, 5(1):47–76, Feb. 1987.

[9] D. E. Culler, R. M. Karp, D. Patterson, A. Sahay, E. E. Santos, K. E. Schauser, R. Subramonian, and T. von Eicken. LogP: A practical model of parallel computation. *Commun. ACM*, 39(11):78–85, Nov. 1996.

[10] S. E. Deering. RFC 1112: Host extensions for IP multicasting, Aug. 1989.

[11] C. Dwork, M. Herlihy, and O. Waarts. Contention in shared memory algorithms. *J. ACM*, 44(6):779–805, Nov. 1997.

[12] R. Guerraoui and A. Schiper. Total order multicast to multiple groups. In *Proc. 17th Int'l Conf. on Distributed Computing Systems (ICDCS)*, pages 578–585, May 1997.

[13] V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. In S. Mullender, editor, *Distributed Systems*, chapter 5, pages 97–146. Second edition, 1993.

[14] A. Heddaya and K. Park. Congestion control for asynchronous parallel computing on workstation networks. *Parallel Computing*, 23(13):1855–1875, Dec. 1997.

[15] J.-H. Huang, C.-C. Yang, and N.-C. Fang. A novel congestion control mechanism for multicast real-time connections. *Computer Communications*, 22:56–72, 1999.

[16] M. F. Kaashoek and A. S. Tanenbaum. Fault tolerance using group communication. *ACM Operating Systems Review*, 25(2):71–74, Apr. 1991.

[17] E. V. Krishnamurthy. Complexity issues in parallel and distributed computing. In A. Y. H. Zomaya, editor, *Parallel & Distributed Computing Handbook*, pages 89–126. 1996.

[18] L. I. Kronsjö. PRAM models. In A. Y. H. Zomaya, editor, *Parallel & Distributed Computing Handbook*, pages 163–191. McGraw-Hill, 1996.

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

[20] C.-C. Lim, L.-J. Yao, and W. Zhao. A comparative study of three token ring protocols for real-time communications. In *Proc. 11th Int'l Conf. on Distributed Computing Systems (ICDCS)*, pages 308–317, May 1991.

[21] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[22] B. Rajagopalan and P. McKinley. A token-based protocol for reliable, ordered multicast communication. In *Proc. 8th Symp. on Reliable Distributed Systems (SRDS)*, pages 84–93, Oct. 1989.

[23] M. Raynal. *Networks and Distributed Computation*. MIT Press, 1988.

[24] A. Schiper. Early consensus in an asynchronous system with a weak failure detector. *Distributed Computing*, 10(3):149–157, 1997.

[25] N. Sergent. *Soft Real-Time Analysis of Asynchronous Agreement Algorithms Using Petri Nets*. PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland, 1998.

[26] D. D. Sleator and R. E. Tarjan. Amortised efficiency of list update and paging rules. *Commun. ACM*, 28(2):202–208, Feb. 1985.

[27] K. Tindell, A. Burns, and A. J. Wellings. Analysis of hard real-time communications. *Real-Time Systems*, 9(2):147–171, Sept. 1995.

[28] L. G. Valiant. A bridging model for parallel architectures. *Commun. ACM*, 33(8):103–111, Aug. 1990.

— Appendix —

A Latency Analysis of the Total Order Broadcast Algorithms

In this section, we describe the latency analysis of some of the algorithms described in Sect. 5.1. We start by the simple analysis of the Token algorithm, then we describe part of the analysis of Skeen's algorithm.

A.1 The Token Algorithm

This algorithm is rather simple to analyze. The reason is that only the process that has the token at the given moment may send messages, i.e., there is no contention on the network (or at least, only contention introduced by only *one* of the processes).

The execution of the Token algorithm can be seen as a sequence of phases. A phase starts when a process p_i receives the token and ends when the next process p_j ($j = (i+1) \bmod n$)) receives the token. There are two types of phases: in the first type, p_i broadcasts a message after receiving the token; in the second type, it does not. We now analyze the latency of these phases.

- If p_i does not want to broadcast, the token takes $2\lambda + 1$ time unit to travel from p_i to p_j. (CPU_i sends: λ, the network transmits: 1, CPU_j receives: λ time units.)
- If p_i also broadcasts, it receives the token, sends the broadcast message to all other processes, then passes on the token. The token now takes $1 + 2\lambda + \max(1, \lambda)(n-1)$ time units from p_i to p_j.

We now use the latencies of the phases to compute the overall latency of the Token algorithm. Let p_1 be the process that initiates A-Broadcast(m); m is the broadcast message.

We do not know where the token is at the time of A-Broadcast(m), since the token circulates at all times, even if there is no message to broadcast. However, we can compute how much p_1 has to wait to obtain the token *on the average*. One round-trip takes $n(2\lambda+1)$ time units, hence, on average, p_1 has to wait $\frac{n}{2}(2\lambda+1)$ time to get the token. The next step is that p_1 broadcasts the message and passes on the token ($1 + 2\lambda + \max(1, \lambda)(n-1)$ time units). Let us now count how many times the token is passed on until the last delivery. The token reaches $p_2, \ldots, p_n$, then again $p_1, \ldots, p_{n-1}$ (see Fig. 3(c)); altogether, it is passed on $2(n-1)$ times.[7] All this yields the following latency for the Token algorithm:

$$\begin{aligned}\text{Latency}(\text{Token})(n,\lambda) &= \left(\frac{n}{2}(2\lambda+1)\right) + (1 + 2\lambda + \max(1,\lambda)(n-1)) + (2(n-1)\cdot(2\lambda+1)) = \\ &= (2.5n-1)(2\lambda+1) + \max(1,\lambda)(n-1)\end{aligned}$$

[7] At the end of the first round-trip, the token has the information that all have received m. Knowing this, processes can safely deliver m in the second round-trip.

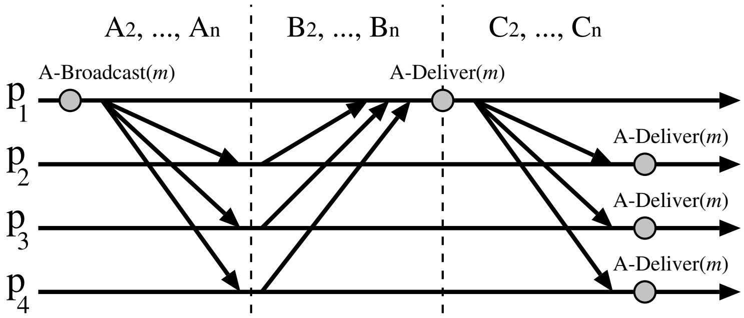


Figure 6: The communication pattern of Skeen's algorithm.

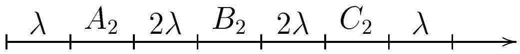


Figure 7: Network utilization graph, showing how the network resource is utilized at any point in time.

A.2 Skeen's Algorithm

Let us first label the messages exchanged in one execution (see Fig. 6): A_i ($i = 2, \ldots, n$) denotes the message sent from p_1 to p_i. The acknowledgment from p_i is labeled B_i; C_i denotes the message that p_1 sends to p_i after receiving all acknowledgments.

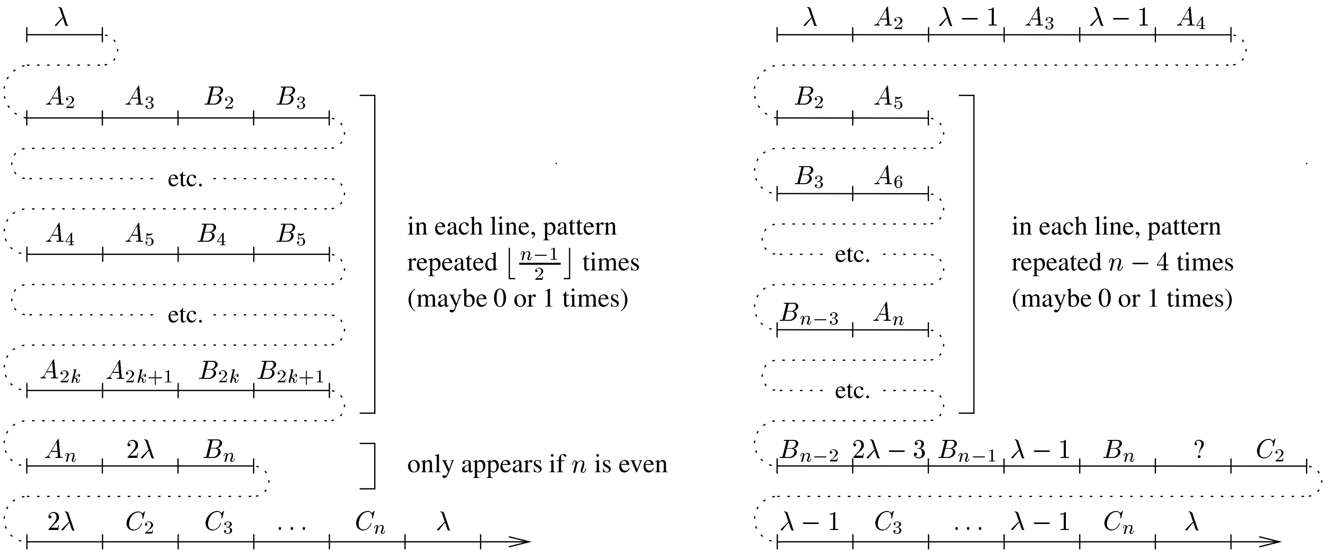
We now introduce a special type of graph to visualize algorithm executions, the *network utilization graph*. The graph focuses on the activity on the network resource. It is a time axis, divided into time intervals. Each of the intervals either corresponds to the transmission of a message or is an interval when the network is idle. In the former case, the interval has the label of the transmitted message. Such intervals are 1 time unit long, as transmitting a message always takes 1 time unit on the network. In the latter case (when the network is idle) the interval is labeled with the duration of idleness.

Figure 7 shows an example for the network utilization graph, for Skeen's algorithm with $n = 2$ (λ can be anything). CPU_1 sends A_2 during the first interval (λ time units), then message A_2 is transmitted. CPU_2 receives A_2 and sends B_2 during the third interval ($\lambda + \lambda$ time units), then B_2 is transmitted. Now, CPU_1 receives B_2 and sends C_2 (2λ time units), and C_2 is transmitted. Finally, CPU_1 receives C_2 (λ time units).

To analyze the algorithm, we start by examining how the network resource is utilized in executions with different values for λ, e.g., with the aid of network utilization graphs. This yields that we have to distinguish 5 cases where the network utilization is fundamentally different:

$$\lambda < 0.5; \quad 0.5 \leq \lambda < 1; \quad 1 \leq \lambda < 1.5; \quad 1.5 \leq \lambda < 2; \quad \lambda \geq 2$$

We only present the cases $\lambda < 0.5$ and $1.5 \leq \lambda < 2$ in detail here. The other cases are similar to either the one or the other, hence presenting them would offer no great additional benefit to the reader. (However, all cases appear in the results at the end of this section.) Also, rather than proving the results, we present the intuition behind the proof. We believe that this yields a more valuable explanation.



(a) $\lambda < 0.5$ (b) $1.5 \leq \lambda < 2$

Figure 8: Network utilization graph of Skeen's algorithm.

Case $\lambda < 0.5$**.** Figure 8(a) shows the network utilization graph for this case. (The time axis is too long to be represented to one line, so it was cut into smaller pieces which are connected by dotted lines.) We can see that the graph is not fundamentally different for various values of n: the only difference is that the repeating pattern starting with A_2 and the one starting with C_2 appear a different number of times.

Computing the latency metric now only amounts to adding up the time intervals shown in the network utilization graph. We obtain the following result:

$$\text{Latency}(\text{Skeen})(n, \lambda) = 3(n-1) + 4\lambda + \begin{cases} 2\lambda & \text{if } n \equiv 0 \pmod 2 \\ 0 & \text{otherwise} \end{cases} \qquad \text{(case } \lambda < 0.5\text{)}$$

Case $1.5 \leq \lambda < 2$**.** Figure 8(b) shows the network utilization graph for this case. (We assume that $n \geq 4$. Executions with $n = 2$ and $n = 3$ have to be analyzed separately.) The graph shows an unknown value ("?") after the transmission of B_n. The reason is that CPU_1 might not have finished receiving $B_2, \dots, B_{n-1}$ at this point, and C_2 is only sent after receiving $B_2, \dots, B_n$. We now proceed with examining the utilization of CPU_1.

Let us define the *work function* W(t) of CPU_1. W(t) tells how much work CPU_1 still has to do to empty its sending and receiving queue, at time t. (Time 0 is the start of the algorithm.) W(t) is thus the number of messages in these two queues times λ, plus the time needed to finish processing the message currently on CPU_1. W(t) is important for us because it gives information about the activity of CPU_1: $W(t) = 0$ if and only if CPU_1 is idle at time t.

We claim that CPU_1 is not idle between time 0 and the end of the transmission of B_n. Let $t(M)$ denote the time when the transmission of message M finishes. An equivalent statement is then that $W(t) > 0$ for all $0 \leq t \leq t(B_n)$. We now prove this equivalent statement.

W(0) is $(n-1)\lambda$, as initially there are $n-1$ messages in the sending queue. W(t) decreases linearly with a slope of -1. When CPU_1 is transmitted a message, W(t) increases by λ. These observations imply that $W(t)$ has local minima at $t(B_2)-0, t(B_3)-0, \ldots, t(B_n)-0$. Therefore we only need to check $W(t) > 0$ for these t values. This computation is best done using the network utilization graph; the result is

$$\begin{aligned} W(t(B_i)-0) &= (n-1)\lambda-(3\lambda+2)+(i-2)\cdot(\lambda-2) \quad (\text{where } 2 \leq i \leq n) \\ &= (2n-6)\lambda-2n+2 \\ &\geq (2n-6)\cdot 1.5-2n+2 = n-7 \end{aligned}$$

All $W(t(B_i)-0)$ values are non-negative if $n \geq 7$. Thus we finished proving that $W(t) > 0$ for $0 \leq t < t(B_n)$ if $n \geq 7$. (The cases $n < 7$ should be examined separately.) Hence we know that CPU_1 works without interruption until it processes all messages up to B_n. This takes $2(n-1)\lambda$ time units, as we have $2(n-1)$ messages until this point. Sending C_2 takes λ time units, thus the transmission of C_2 starts at time $(2n-1)\lambda$. One can then easily compute the latency of the remaining part of the execution based on the network utilization graph. The overall latency of the algorithm is as follows:

$$\mathrm{Latency}(\mathrm{Skeen})(n,\lambda) = (3n-2)\lambda+1 \quad (\text{case } 1.5 \leq \lambda < 2 \text{ and } n \geq 7)$$

Final result. The latency analysis of all cases leads to the following result:

$\mathrm{Latency}(\mathrm{Skeen})(n,\lambda) =$

$$\begin{cases} 3(n-1)+4\lambda+\left\{\begin{array}{ll} 2\lambda & \text{if } (\lambda < 0.5 \text{ and } n \equiv 0 \pmod 2) \\ & \text{or } (\lambda \geq 0.5 \text{ and } n \equiv 2 \pmod 3) \\ 2\lambda-1 & \text{if } (\lambda \geq 0.5 \text{ and } n \equiv 0 \pmod 3) \\ 0 & \text{otherwise} \end{array}\right\} & \text{if } \lambda < 1 \\ (3n-2)\lambda+1+ & \\ +\left\{\begin{array}{ll} 2+(4-n)\lambda & \text{if } n \leq 4 \\ \max(0, 4-2\lambda) & \text{if } n = 5 \\ 0 & \text{if } n \geq 6 \text{ and } \lambda \geq 1.5 \\ \max\left(0, \begin{array}{l} \left\{\begin{array}{ll} \lambda+1 & \text{if } n \equiv 0 \pmod 4 \\ 2 & \text{if } n \equiv 1 \pmod 4 \\ 3 & \text{otherwise} \end{array}\right\} \\ -(n-4)(2\lambda-2) \end{array}\right) & \text{if } n \geq 6 \text{ and } \lambda < 1.5 \end{array}\right\} & \text{if } \lambda \geq 1 \end{cases}$$

The expression looks complicated. Fortunately, we do not need to use it in this form: as we normally use the latency metric for comparing algorithms, it is perfectly enough if we can give bounds for the expression. The resulting simplified form is shown below. (We exploit the fact that the complicated parts in curly brackets are small compared to the whole expression.)

$\mathrm{Latency}(\mathrm{Skeen})(n,\lambda) =$

$$\begin{array}{ll} 3(n-1)+4\lambda+c_1 \text{ where } 0 \leq c_1 \leq 2\lambda & \text{if } \lambda < 1 \\ (3n-2)\lambda+1+c_2 \text{ where } 0 \leq c_2 \leq 2+2\lambda & \text{if } \lambda \geq 1 \end{array}$$

B Throughput Analysis of the Atomic Broadcast Algorithms

In this section, we compute the throughput metric (see Sect. 4.2) for the four Atomic Broadcast algorithms of Sect. 5.1.

The steps are the same for each algorithm $\mathcal{A}$. First we determine how the algorithm behaves under high load (as defined in Sect. 5.3). We then count the number of send and receive operations on all CPUs and the number of transmissions on the network. This yields the utilization times ($T_r(n, \lambda)$) for each resource r (taken from the set of all resources $\mathcal{R}_n$). The metric is then computed using the formula

$$\mathrm{Thput}(\mathcal{A})(n, \lambda) = \frac{1}{\max_{r \in \mathcal{R}_n} T_r(n, \lambda)}$$

Lamport Because of the high load on the system, "null messages" are not needed. The only messages generated per A-Broadcast(m) event are those of the initial broadcast, altogether $n-1$ messages.

Recall that every process generates A-Broadcast(m) events, thus the sender process is not always the same. However, each process becomes a sender equally often. Therefore we must take the *average utilization time* of the CPUs into account.

The remaining steps of the evaluation are summarized in the following table:

Resource r	**Activity of r**	$\mathbf{T_r(n, \lambda)}$
CPU (sender)	$(n-1)$ send	$(n-1)\lambda$
CPU (recipients)	1 receive	λ
CPU (average)	–	$(n-1)\frac{2\lambda}{n}$
network	$n-1$ transmissions	$n-1$

The resulting throughput is

$$\mathrm{Thput}(\mathrm{Lamport})(n, \lambda) = \frac{1}{(n-1) \cdot \max\left(1, \frac{2\lambda}{n}\right)}$$

Skeen The algorithm behaves the same under high and low load. Each A-Broadcast(m) results in the messages depicted in Fig. 3(b). Just as with Lamport, we have to take the average utilization time of the CPUs into account.

Resource r	**Activity of r**	$\mathbf{T_r(n, \lambda)}$
CPU (sender)	$2(n-1)$ send, $n-1$ receive	$3(n-1)\lambda$
CPU (recipients)	1 send, 2 receive	3λ
CPU (average)	–	$3(n-1)\frac{2\lambda}{n}$
network	$3(n-1)$ transmissions	$3(n-1)$

The resulting throughput is

$$\mathrm{Thput}(\mathrm{Skeen})(n, \lambda) = \frac{1}{3(n-1) \cdot \max\left(1, \frac{2\lambda}{n}\right)}$$

Token Each process obtains the token, sends k broadcasts and passes on the token. (k is a parameter of the Token algorithm. Its purpose is to put a limit on the token holding time.) Let $k = 1$ for simplicity. Then there are n messages generated per request. Once again, we take the average utilization time of CPUs into account.

Resource r	**Activity of r**	$\mathbf{T_r(n, \lambda)}$
CPU (sender)	n send	$n\lambda$
CPU (recipient following the sender)	2 receive	2λ
CPU (other recipients)	1 receive	λ
CPU (average)	–	2λ
network	n transmissions	n

We obtain the following throughput:

$$\mathrm{Thput}(\mathrm{Token})(n, \lambda) = \frac{1}{n \cdot \max\left(1, \frac{2\lambda}{n}\right)}$$

Sequencer The utilization of CPUs is different in this algorithm. As before, every process becomes the sender equally often, but the sequencer is always the same process. We have to take the (average) utilization time of the sequencer and that of the other processes into account.

Resource r	**Activity of r**	$\mathbf{T_r(n, \lambda)}$
CPU (sequencer and sender)	$n - 1$ send	$(n-1)\lambda$
CPU (sequencer, if different from sender)	1 receive, $n - 1$ send	$n\lambda$
CPU (sender, if different from sequencer)	1 send, 1 receive	2λ
CPU (recipients)	1 receive	λ
CPU (sequencer, average)	–	$\left(n - \frac{1}{n}\right)\lambda$
CPU (others, average)	–	$\left(1 + \frac{1}{n}\right)\lambda$
network (average)	n (or $n - 1$) transmissions	$n - \frac{1}{n}$

The resulting throughput is

$$\mathrm{Thput}(\mathrm{Sequencer})(n, \lambda) = \frac{1}{\left(n - \frac{1}{n}\right) \cdot \max(1, \lambda)}$$

C Analysis for Broadcast Networks

In this section, we analyze the Atomic Broadcast algorithms of Sect. 5.1 using the metrics defined for broadcast networks. The analyses are essentially similar to the analyses in the case of point-to-point networks, therefore we choose to present them in a less detailed way. They are simpler, though, as there are fewer messages and less contention in the algorithms under investigation.

C.1 Lamport

Latency

$$\begin{aligned}\text{Latency}_{br}(\text{Lamport})(n,\lambda) &= \lambda + 1 + 2\lambda + n - 1 + \lambda \\ &= 4\lambda + n\end{aligned}$$

Throughput

Resource r	**Activity of r**	$\mathbf{T_r(n,\lambda)}$
CPU (sender)	1 send	λ
CPU (recipients)	1 receive	λ
CPU (average)	-	λ
network	1 transmission	1

The resulting throughput is

$$\text{Thput}_{br}(\text{Lamport})(n,\lambda) = \frac{1}{\max(1,\lambda)}$$

C.2 Skeen

Latency

$$\begin{aligned}\text{Latency}_{br}(\text{Skeen})(n,\lambda) &= \lambda + 1 + 2\lambda + 1 + (-n2)\max(1,\lambda) + 2\lambda + 1 + \lambda \\ &= 6\lambda + 3 + (n-2)\max(1,\lambda)\end{aligned}$$

Throughput

Resource r	**Activity of r**	$\mathbf{T_r(n,\lambda)}$
CPU (sender)	2 send, $(n-1)$ receive	$(n+1)\lambda$
CPU (recipients)	1 send, 2 receive	3λ
CPU (average)	-	$\frac{4n-2}{n}\lambda$
network	$n+1$ transmission	$n+1$

The resulting throughput is

$$\text{Thput}_{br}(\text{Skeen})(n,\lambda) = \frac{1}{\max\left((n+1), \frac{4n+1}{n}\lambda\right)}$$

C.3 Token

Latency

$$\begin{aligned}\text{Latency}_{br}(\text{Token})(n,\lambda) &= \frac{n}{2}(2\lambda+1) + \max(\lambda,1) + (2n-1)(2\lambda+1) \\ &= \left(\frac{5n}{2} - 1\right)(2\lambda+1) + \max(\lambda,1)\end{aligned}$$

Throughput

Resource r	Activity of r	$\mathbf{T_r(n, \lambda)}$
CPU (sender)	2 send	2λ
CPU (recipient following sender)	2 receive	2
CPU (other recipients)	1 receive	1
CPU (average)	-	$\frac{n+2}{n}\lambda$
network	2 transmission	2

The resulting throughput is

$$\mathrm{Thput}_{br}(\mathrm{Token})(n, \lambda) = \frac{1}{\max\left(2, \frac{n+2}{n}\lambda\right)}$$

C.4 Sequencer

Latency

$$\begin{aligned}\mathrm{Latency}_{br}(\mathrm{Sequencer})(n, \lambda) &= \lambda + 1 + 2\lambda + 1 + \lambda \\ &= 4\lambda + 2\end{aligned}$$

Throughput

	Resource r	Activity of r	$\mathbf{T_r(n, \lambda)}$
seq $\neq$ sender	CPU (sender)	1 send, 1 receive	2λ
	CPU (sequencer)	1 send, 1 receive	2λ
	CPU (recipients)	1 receive	λ
	network	2 transmission	2
seq $=$ sender	CPU (sender, sequencer)	1	λ
	CPU (recipients)	1 receive	λ
	network	1 transmission	1
average	CPU (sequencer)	-	$\frac{2n-1}{n}\lambda$
	CPU (recipients)	-	$\frac{n+1}{n}\lambda$
	network	-	$\frac{2n-1}{n}$

The resulting throughput is

$$\mathrm{Thput}_{br}(\mathrm{Sequencer})(n, \lambda) = \frac{n}{(2n-1)\max(1, \lambda)}$$

The average for CPU(recipients) can be ignored because it is larger than CPU(coordinator) only if $n < 2$. Then, this does not make sense because there can be no distinction between recipient and sequencer in such a case.

C.5 Synopsis

C.5.1 Latency

$\boxed{\text{Sequencer} > \text{Lamport} > \text{Skeen} > \text{Token}}$

C.5.2 Throughput

- $\lambda \leq \begin{cases} \max\left(1, \frac{2n}{2n-1}\right) & \text{if } n > 3 \\ \max\left(1, \frac{2n}{n+2}\right) & \text{otherwise} \end{cases}$ $\boxed{\text{Lamport} < \text{Sequencer} \leq \text{Token} < \text{Skeen}}$
- otherwise $\boxed{\text{Lamport} < \text{Token} < \text{Sequencer} < \text{Skeen}}$