Evaluation of Message Delay Correlation in Distributed Systems

Daniel Albeseder[1]*

[1]Technische Universität Wien, Embedded Computing Systems Group E182/2
Treitlstrasse 3, A-1040 Vienna (Austria), da@ecs.tuwien.ac.at

Abstract — *Partially synchronous computational models fall in between synchronous models, which are demanding in terms of requirements but admit solutions to most of the important fault-tolerant distributed computing problems, and the purely asynchronous model, where this is not the case. The Θ-Model is a recently presented partially synchronous model close to pure asynchrony. It just assumes a bound on the ratio of maximum vs. minimum end-to-end delays of messages simultaneously in transit. This paper provides experimental evidence for the assumed correlation of end-to-end delays in some common type of distributed systems: Using a comprehensive custom evaluation framework, we measured the end-to-end delays in a simple clock synchronization algorithm running on a Fast-Ethernet network connecting Linux workstations. Our results reveal a significant correlation between maximum and minimum delay under several load conditions, and hence confirm that the bounded ratio assumption of the Θ-Model is well-founded.*

1 Introduction

Today, most real-time system designs are based on synchronous models of computation. Although synchronous solutions are indeed quite easy to understand and to verify, their domain of applicability is restricted due to the required a priori known bounds on computational step time and transmission delays, which are load dependent. Obviously at high network load, the current message delays are significantly larger than the delays which are experienced in a network during situations that are close to idle. As laid out in [1], self organizing solutions that adapt automatically to the current load and thus always perform as good as the underlying system allows is one major goal of next generation embedded systems.

A system model that has at its core a self organizing property is the Θ-Model [2, 3]. Rather than a priori known bounds on maximum and minimum end-to-end delays, the Θ-Model assumes only an a priori known bound on the ratio of all messages that are simultaneously in transit. In addition, it facilitates purely message-driven (clock and timer

*Supported by the FWF-project *Theta* (project no. 17757-N04)

free) algorithms, which are only controlled by message receptions and do not depend on or consist of any timing values. Thus algorithms in the Θ-Model are independent of bounds on the transmission delay, given that the ratio holds. The question explored in this paper is whether this assumption is valid in current distributed systems.

There is an intuitive argument in favor of such a correlation, at least for some classes of distributed systems: In case of cooperative algorithms, where broadcasting is the main communication primitive, the system load is very likely to be equally distributed. If one process suffers from the worst-case load, it is unlikely that some other process (executing the same distributed algorithm) is totally idle at the same time. This has also been confirmed by schedulability analysis in case of shared channel based systems [4]. The present paper provides experimental evidence for this correlation also in a distributed system of switched Ethernet-coupled Linux workstations.

The major advantage of the Θ-Model over time-driven models is its capability for coverage expansion: Consider overloads or other unexpected operating conditions that lead to violations of the a priori upper bound on delays, thereby violating synchronous model assumptions. In case the actual minimum delay also increases, the ratio Θ and hence the Θ-Model may still be valid.

2 Model

Network We survey the Θ-Model which was introduced in [2]: We consider a system of n distributed processors with only one process per processor (in general the model is not limited to this). From the n processes, up to f may behave Byzantine faulty. The network between the processors is modeled as a reliable fully-connected point-to-point network. Note that we will actually use a switched Ethernet for the evaluation. Given our high-performance switches, it is considered to behave like a fully-connected network. We consider two communication primitives for the algorithms: A broadcast primitive, which sends a message to all processes, and a message reception function. Furthermore, we suppose the sender of every message can be unambiguously determined from the network link over which the message was received; i.e., we assume in our evaluation that the sender address of a message cannot be forged. For a real application this can be obtained by message authentication.

Timing The end-to-end delay δ of a message is the time span from the invocation of the broadcast primitive at the sender process at time t_{BE} (called broadcast event) to the completion of message processing at the receiver at time t_{PE} (denoted perception event) cf. Figure 2. Thus δ includes not only the transmission delay but also the computational time to generate and process the message plus sojourn times in waiting queues. A message is called *in transit* at time t if $t_{BE} \leq t < t_{PE}$. In the following definitions, we consider only messages sent and received by correct processes over a correct link. We define $\tau^+(t)$ as the maximum end-to-end delay of messages which are in transit at time t. Respectively, $\tau^-(t)$ denotes the minimum end-to-end delay at time t. The current uncertainty ratio at time t can be obtained by division of these two values: $\vartheta(t) = \tau^+(t)/\tau^-(t)$. Note that $\vartheta(t) \geq 1$ trivially holds for all times t, since $\tau^+(t) \geq \tau^-(t)$.

For describing the time-independent behavior of the system we define the cumulative

uncertainty ratio $\Theta = max_t(\vartheta(t))$ as the maximum of all current uncertainty ratios over all times t. Another ratio can be determined by employing the overall bounds of the end-to-end delays — i.e., $\tau^+ = max_t(\tau^+(t))$ and $\tau^- = min_t(\tau^-(t))$ — to define a pessimistic overall uncertainty ratio $\Theta_{p,o} = \tau^+/\tau^-$. Note that this ratio is not relevant for the timing of our algorithms. We just determine it for the sake of numerical comparison.

Significant Messages Algorithms in the Θ-Model are timer free and message-driven. They usually advance in asynchronous rounds, where every process switches to the next round and broadcasts a message upon reception of the $n - f$-th message of the current round. This implies that the round switching times e_i are typically not determined by the shortest message delay $\tau^-(t)$, but by the time $n - f$ fast messages require for being delivered. Therefore, the delay of the $n - f$-th fastest message is relevant to us, and we denote it by $\delta_r^p(t)$. So the relevant fastest delay is $\tau_r^-(t) = max_p \delta_r^p(t)$ for all processes p which have a round switch at time $t = e_i$. If t is no round-switching time, the relevant fastest delay is defined as $\tau_r^-(t) = \tau_r^-(e_i)$, where $e_{i-1} < t < e_i$. Clearly, e_{i-1} and e_i denote two successive round-switching times, i.e. times where the first correct process switches to round i resp. $i + 1$. For our evaluation, we therefore only have to measure $\tau_r^-(t)$ and $\tau^+(t)$ at round-switching times. If there is no message in transit at time t, $\tau^+(t) = \tau_r^-(t)$ is set by definition. The global minimum of $\tau_r^-(t)$ is defined by $\tau_r^- = min_t(\tau_r^-(t))$. This leads us to the significant uncertainty ratio $\Omega = max_t(\tau^+(t)/\tau_r^-(t))$ and the overall realistic uncertainty ratio $\Theta_{r,o} = \tau^+/\tau_r^-$. For an in-depth description of the Θ-Model and Θ-algorithms consult [5].

3 Clock Synchronization Algorithm

For our evaluation, we employed a simplified version of the consistent broadcast based [6] clock synchronization algorithm from [7, 5]. It maintains an integer-valued clock $C_p(t)$ at any process p that satisfies the usual precision and envelope conditions, for $n \geq 3f + 1$.

In the algorithm of Figure 1, variable k holds the current round number, which represents also the current clock value $C_p(t)$. The algorithm uses messages that carry integers[1] for disseminating the current clock value. After initialization, the algorithm sends an *(init,* 1*)* message to all processes. Depending on how many *(init, k)* messages were already received, one of the two rules may be executed at the receiver. The first rule (line 6) is the *catch-up rule*, which guarantees correct processes that lagged behind the current clock value to catch-up quickly. The second rule (line 12) allows the clock value to advance to round $k + 1$, provided that a sufficiently large quorum of correct processes are already in round k.

Since the algorithm is message driven, it would execute as fast as message arrivals allow. In order to slow down progress and hence reduce processing and network load in systems with small delay × bandwidth product [2], some local delay $D > 0$ can be introduced in line 13 (with the drawback of losing the purely message-driven property).

Theoretical analysis [5] revealed a precision of $\lfloor \Omega + 2 \rfloor$ if $D = 0$. The clock rate (i.e., the number of increments of k per second) is within $[O(1/\tau^+), O(1/\tau_r^-)]$.

[1] In practice, unbounded integers can be approximated by sufficiently large variables.

```
0:  VAR k : integer := 1;
1:
2:  /* Initialization */
3:  send (init, 1) to all [once];
4:
5:  /* catch-up rule */
6:  if received (init, ℓ) from at least f + 1 distinct processes with ℓ ≥ k
7:       → k := ℓ; /* jump to new round */
8:          send (init, k) to all [once];
9:  fi
10:
11: /* advance rule */
12: if received (init, k) from at least n − f distinct processes
13:         delay(D);
14:      → k := k + 1;
15:         send (init, k) to all [once]; /* start next round */
16: fi
```

Figure 1: Clock-Synchronization Algorithm for a process p

4 Evaluation System

The evaluation system consists of a network of several workstations (Pentium 4, 2.4GHz FSB533) running Redhat[2] Linux 7.2 with a modified 2.4.20 kernel. The fully-connected network is emulated by a 100MBit/s switched Fast-Ethernet. In our experiments, one workstation was dedicated to control the test runs and four to seven workstations executed the algorithm.

The kernel needed some adaptations, such that the workstations were suited for running the evaluation and the algorithms. The main improvement compared to a vanilla Linux[3] kernel was the high-resolution kernel timers patch [8], which gives kernel-timers the ability to trigger with a precision about 1μs, while ordinary Linux kernel timers can trigger only by multiples of a tick, which is 10ms for a standard 2.4 kernel. Since responsiveness of the kernel is a major issue, the kernel-preemption patch [9] was used as well.

Moreover, it happens on rare occasions, that a bottom-half[4] of the timer-interrupt is not executed directly after the interrupt service routine and will be delayed until the next tick. This would have severe effects on our delay(D)'s timing. This situation was circumvented by increasing the priority of the kernel thread `ksoftirqd` — which is responsible for executing the bottom-halves — to the maximum real-time priority. This ensures that even if the bottom-half is not executed directly after the interrupt service routine, `ksoftirqd` is scheduled immediately afterwards.

One of the most crucial parts for Θ-algorithms is scheduling. This can be divided

[2] `http://www.redhat.com`

[3] Vanilla Kernel denotes the original Kernel released by Linus Torvalds, without any patches applied (`http://www.kernel.org`).

[4] Linux interrupt drivers consists of two parts, the top-half, which is executed immediately in the low-level interrupt handler, and the bottom-half, which is hence deferred and executed later in kernel context. So interrupt handlers only block the system for a small amount of time, but larger computations initiated by an interrupt can be done quite fast.

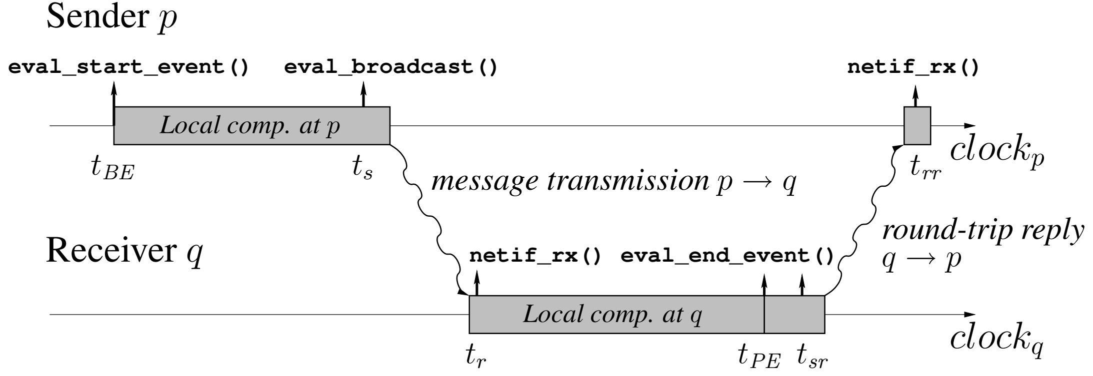

Figure 2: One message round-trip transparently measured by the EvalAPI function calls.

into message and task scheduling. For message scheduling, Linux uses FIFO-queues by default. However, other message scheduling algorithms can be employed, but just for incoming messages, and only at IP-level. So the lower network levels still use FIFO-queues. For outgoing messages there is no scheduling implementation in Linux by default so only FIFO-queuing is used here. For task scheduling, Linux implements the POSIX1.b real-time scheduling policies. These define fixed preemptive real-time priorities with scheduling types FIFO and RR (round robin) which can be used for employing head-of-line scheduling [10]. By using one of these real-time priorities one can guarantee task invocation (preemption of a lower priority task), whenever a task with a higher priority than the current running task becomes ready.

The evaluation system [11, 12] consists of the evaluation software `evalpsa` and the controlling software `autopsa`. The controlling software starts and ends test runs and sends load change requests to the evaluation software. The evaluation software consists of four tasks. The first is the algorithm task itself, which contains the algorithm to be evaluated, linked together with the `EvalAPI`. The latter provides the functions in particular instrumented versions of message broadcast and receive primitives necessary to implement and evaluate Θ-algorithms. Figure 2 shows a round-trip measurement by using `EvalAPI` functions to track the end-to-end delays. The `netif_rx` function is a Linux kernel internal function, which is called by the network device driver whenever a message was received. This function creates a time stamp which we are using. The other functions are `EvalAPI` functions, which are used by our algorithm implementation.

The second task is the control task, which starts and stops the algorithm task, and also changes load settings over time. The remaining two tasks are dedicated to generate network resp. processor load. For generating low level processor load, a custom kernel-module was used, which generates timer-interrupt load with an adjustable period. Besides this kernel-module, the processor-load task generates I/O load by writing data onto the hard-disk. Finally the netload task simply sends load messages to every processor in a load dependent interval. Suitable real-time priorities were assigned to each task. Since a very busy algorithm task with top priority could not be aborted, the control task was provided with the top priority to prevent a system-lock. This created some additional scheduling latencies, which had only negligible small influence on our results.

5 Measurements

Uncertainty Ratio Calculation NTP [13] provides accuracy of clock synchronization which is not sufficient to do one-way measurements by comparing local send time at the sender to local receive time at the receiver. Thus round-trip delay measurements were used. This was achieved by assuming that an algorithm message needs the same transmission time as a round-trip reply, since both messages have the same size. The end-to-end delay is then computed by adding the local computation times before the send, after the receive and the average of both message delays.

Due to lack of global time, an artificial timebase had to be used to determine which messages were in transit at any particular instant. The clock drifts, which can occur during a single round-trip delay, were assumed negligible; clock drifts during the whole run were considered, however. One processor clock was chosen to be the artificial global time base, and all other clocks were aligned to it by calculating the time precision of all perception events compared to the time base of the chosen processor. This was also based on the assumption that algorithm messages and round-trip reply messages need the same transmission time. These precision values were used for determining the global time of all events on all processors by using the precision value whose time had the smallest absolute difference on the local time scale.

At evaluation start, caching and swapping effects could occur. This, and uneven termination of the algorithm at the end of the test run results in invalid measurements. So the first and last values were not used for data analysis.

Goals The main goal was to verify whether there is a correlation between end-to-end delay bounds in various load scenarios. Moreover, we wanted to get an idea of the order of magnitude of Ω. Thus, first tests with constant system loads were executed. Three scenarios were used:

- pure network load
- pure processor load
- combined network and processor load

The next step was to observe the system under varying system loads; tests were conducted with increasing and decreasing loads. Finally, fast load jumps were induced by changing the load from 10% to higher values (50% to 90%) and vice versa every 10ms in order to find out whether fast load changes could result in situations where one process already suffers from the increased load, resulting in higher end-to-end delays, and another one does not. This could lead to larger values of Ω and it was to be examined whether the worst-case bound $\Theta_{r,o}$ is reached in such cases.

Besides the magnitude of Ω, the relation to the overall uncertainty ratios were of major interest. In the following we express these relations via correlation factors $g = \Theta_{p,o}/\Omega$ and $g_r = \Theta_{r,o}/\Omega$.

Test Scenarios Most tests were conducted by using four computers executing the algorithm of Figure 1 with $n = 4$ and $f = 1$; we did, however, refrain from actually injecting

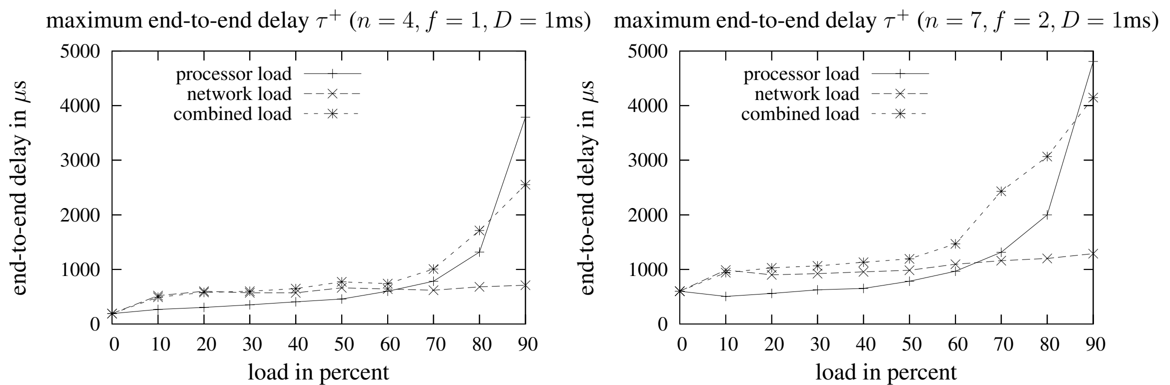

Figure 3: Maximum end-to-end delays τ^+ over five test-runs each

failures. Some tests were repeatedly conducted with $n = 7$ and $f = 2$ to determine the influence the number of computers and number of allowed failures have on the system parameters.

The inter-round delay D was set to 1ms, 100μs and 0s respectively. The constant load and load-jump tests ran for 30 seconds, while the increasing/decreasing load settings used longer runtimes.

The real-time priorities of the tasks were in the following order (from high to low): control task, algorithm task, netload task, cpuload task. Some tests used normal Linux scheduling without real-time priorities for comparison.

6 Results

Since we used only one process per processor, the value n corresponds to the number of workstations in our evaluation. Thus, whenever there is no further notice about the number of processors n and the delay D, the following values are assumed: $n = 4$ and $D =$ 1ms.

In the following figures the left sides represents results for $n = 4$ while the right sides shows results for $n = 7$.

6.1 End-to-End Delay Comparison

Figure 3 shows a not very surprising correlation between the load and τ^+. Figure 4 is more interesting, because some correlation between τ_r^- and the load can be observed. (In contrast, τ^- was about 23μs for the most cases, and only increased slightly for very high load settings.)

We believe the peak for 70% network load is caused by the special network stack implementation of Linux, which favors throughput over transmission delay by switching into polling mode.

Note that the performance of our algorithm does not depend upon the ratio $\Theta_{r,o}$ of τ^+ during high load and τ^- during low load, but rather on $\Omega = max_t(\tau^+(t)/\tau_r^-(t))$. Values for Ω are given in the following section. A comparison of $\Theta_{r,o}$ and Ω can be found in Section 6.3.

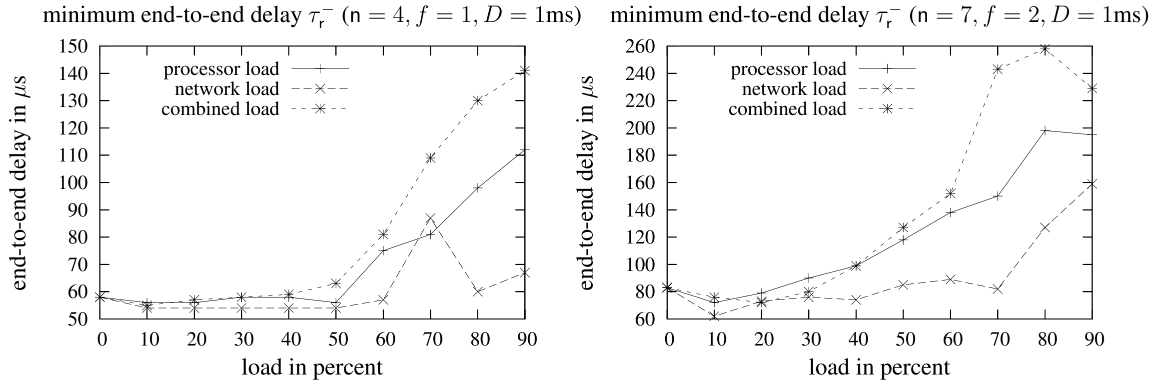

Figure 4: Minimum end-to-end delays τ_r^- over five test-runs each

6.2 Values of Ω

Our main interest was focused on experimental values for the significant uncertainty ratio Ω. Experimental values are given in Figure 5.

load setting	test setting	0%	10%	30%	50%	70%	90%
pure network load	$n = 4, f = 1$	2.72	7.05	8.60	8.78	4.81	8.22
	$n = 7, f = 2$	4.52	8.60	9.78	8.47	8.91	5.23
pure processor load	$n = 4, f = 1$	2.72	3.74	4.29	5.11	6.97	18.47
	$n = 7, f = 2$	4.52	4.43	4.74	5.45	6.56	11.44
combined load	$n = 4, f = 1$	2.72	7.98	7.97	10.15	5.71	10.85
	$n = 7, f = 2$	4.52	8.67	10.94	7.88	5.98	8.66

Figure 5: Maximum Ω over five evaluation runs for $D = 1$ms

The measurements indicate that only very high processor load — i.e., interrupt load — really increases Ω. For $D \geq 100\mu$s using head-of-line scheduling, Ω was below 12 in all other load scenarios. However this does not mean that the assumptions of the Θ-Model are violated for the very high processor load case. It only indicates that the system would perform worse if these load scenarios are considered to be allowed. By contrast, normal Linux scheduling resulted in Ω values above 1800 in some cases. For $D = 0$, Ω was larger than the values above. For example, for no load with $n = 4, f = 1$, it was 16.38, for no load with $n = 7, f = 2$ it was 21.49, and climbed up to 69 in the worst scenario. These experiments also showed some correlation between Ω and the load, and g_r was above 1 (compare Section 6.3). However, a single test-run for $D = 0, n = 7, f = 2$ at 30% processor load did protrude with values for $\tau^+ = 71870\mu$s and $\Omega = 789.8$. It is still not clear what affects led to this big end-to-end delay while using head-of-line scheduling.

Further tests revealed that network load does not influence Ω in a linear way. There are areas of network load where Ω is much lower than usual. This seems to be caused by higher τ_r^- values, which are caused by the Linux network stack implementation as described in Section 6.1.

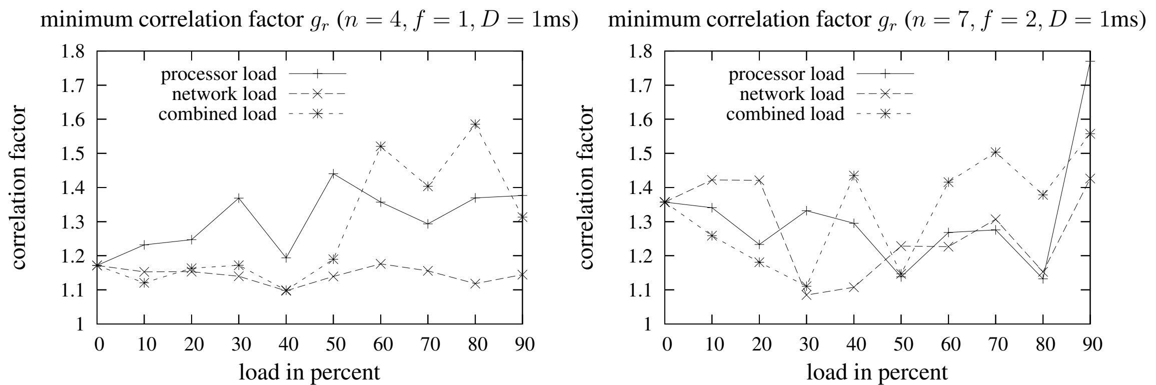

Figure 6: Minimum g_r over five test-runs each

6.3 Correlation Factor

The correlation factor g_r expresses the correlation between τ^+ and τ_r^- for current situations. For load scenarios, where the correlation between these two end-to-end delays to the current load is high, the correlation factor will be also higher than for situations where τ_r^- only increases slightly with the load. In Figure 6 the correlation factor is shown for the constant load cases.

Since both, the long delays and the short ones, are load-dependent (cf. Section 6.1), the most pessimistic uncertainty ratio is the ratio between high load τ^+ and low load τ_r^-. For increasing resp. decreasing load runs, these correlation factors $g_r = \Theta_{r,o}/\Omega$ were as expected greater than in the constant load case. Tangible values for processor load 0%–90% were $g_r > 3.6$ and for network load 0%–99% $g_r > 1.4$.

6.4 Load Jumps

Looking at the load jump measurements, it can be seen that Ω was larger on average. The correlation factors g_r were smaller for most load-jump cases, but always stayed significantly above 1. This shows that the τ^+ vs. τ_r^- correlation exists at least in our setting.

6.5 Histograms of End-to-End Delays and Ω

Formal analysis [5] revealed that not all messages have to be considered when estimating the performance of an algorithm: Just the relation between the duration of messages which actually contribute to a round-switch is significant for the performance of the algorithm. So the end-to-end delays $\delta_{k,x}^p$ for all rounds k and processes p were examined for every round-switching time $e(p,k)$ where $\delta_{k,x}^p$ denotes the x-th fastest message which contributes to a round k switch at process p.

In Figure 7, histograms h_x of the set of $\delta_{k,x}^p$ for all processes p and rounds k for no load and a 70% combined load run are given. $x = 0$ denotes all messages which did not contribute to any round-switch. As you can notice, the histograms h_1 are very different to the other ones. This is due to the fact that in most cases values in h_1 represent self-receptions, where the queuing in the network and in the network driver is bypassed, and there is no real transmission delay. However, the internal processing of self-receptions

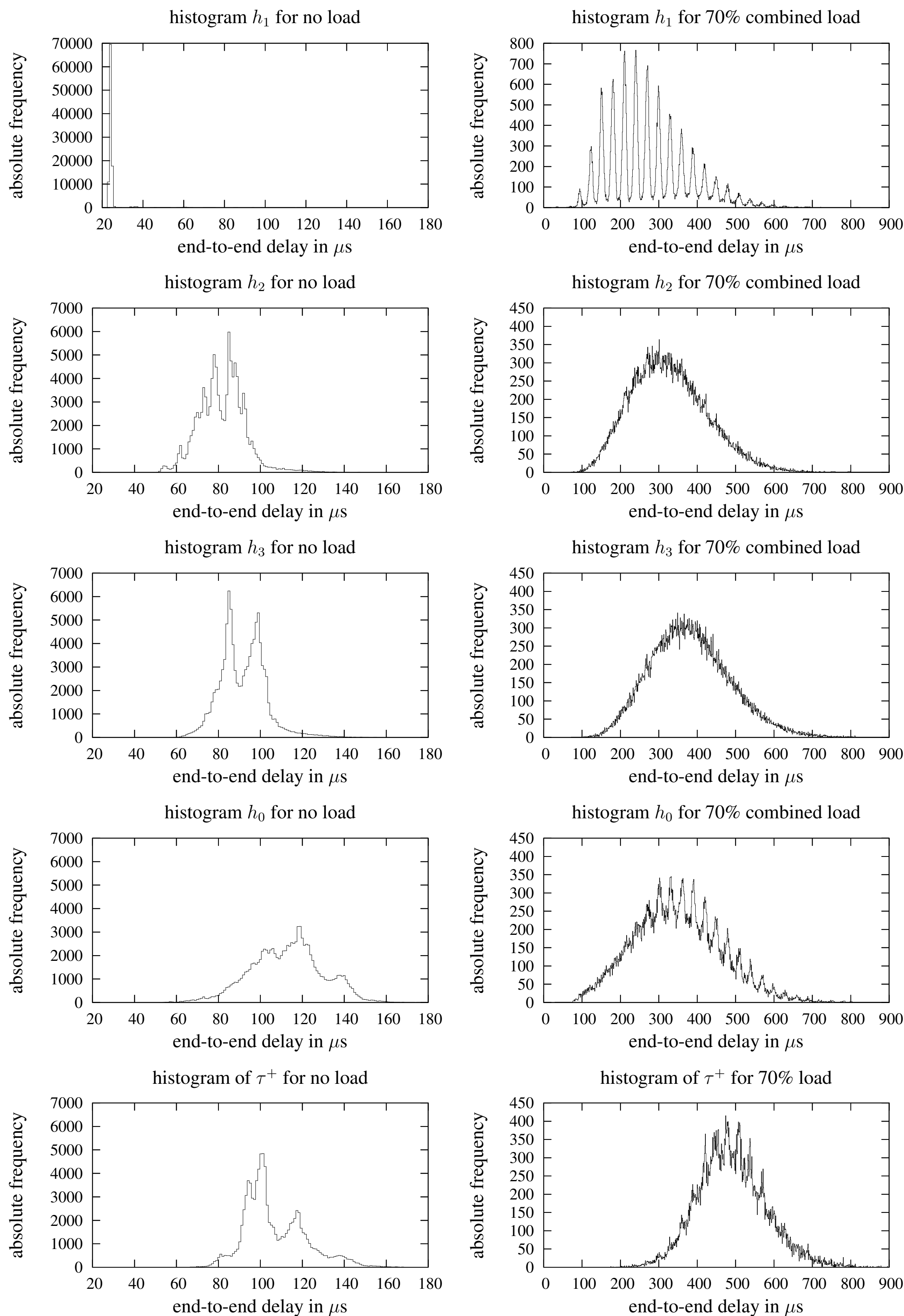

Figure 7: End-to-end delays ordered by speed with bin width 1μs

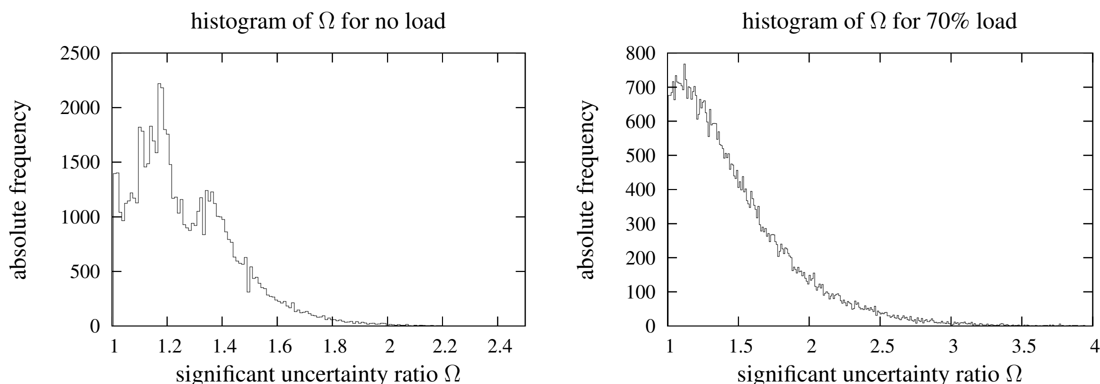

Figure 8: Histogram for $\Omega > 1$ with bin width 0.01

also needs time, thus self-receptions can be delayed by processor and network load. In the 70% load case, the distance between the peaks in h_1 matches the timing delay of the interrupt load generator.

On the other hand h_2 and h_3 look quite similar, but h_3 has an higher average than h_2. The τ^+ histograms show the frequency of τ^+ values for all round-switching times e.

Finally Figure 8 shows the histograms of the Ω values above 1. Since $\Omega = 1$ happens on many round-switches (31971 times for no load, which is 33% of all events, and 21859 times, which is 31%, for 70% load) this would lead to a bad scaling of the histogram, so the peak for $\Omega = 1$ was omitted. Since $\Omega \geq 1$ only this peak is missing inside the histogram.

7 Future Work

Although the results of the measurements were promising in general, there are some shortcomings in the measurement method: Due to the round-trip measurement of the end-to-end delays, the reply messages affect the algorithm and its timing properties. Moreover, the calculated end-to-end delay is in fact the average of two real end-to-end delays. Thus some peaks may average out. To eliminate those effects in future experiments, a low level clock synchronization hardware developed in the course of our SynUTC-project [14] should be used for one-way delay measurements.

It is still not clear whether and how the occurrence of process and link failures affect our results. Conducting failure injection experiments using `evalpsa` is hence mandatory.

A *general shortcoming* of experimental evaluation is limited control over the system, in particular, over message delays and arrival patterns. Consequently, our experiments only allow us to conclude that the Θ-assumption makes sense for some particular operating conditions. In order to improve on this, we will build a suitable simulation framework, which will give us sufficient control over the system behavior.

8 Conclusion

This paper reports on an experimental evaluation of the correlation between maximum and minimum end-to-end delays in a distributed system of Linux workstations connected via

a switched Ethernet network. A simple clock synchronization algorithm for the Θ-Model was implemented via tasks using high real-time priority, and the correlation between minimum and maximum end-to-end delay was noticeable for every load scenario considered, although the actual value of the ratio was not the same in all test runs. This gives some evidence that the stipulated coverage expansion property [15] of the Θ-Model indeed holds true.

References

[1] Wilfried Elmenreich. Intelligent methods for embedded systems. In *Proceedings of the First Workshop on Intelligent Solutions for Embedded Systems*, pages 3–11, Vienna, Austria, June 2003.

[2] Gérard Le Lann and Ulrich Schmid. How to implement a timer-free perfect failure detector in partially synchronous systems. Technical Report 183/1-127, Department of Automation, Technische Universität Wien, January 2003.

[3] Josef Widder, Gérard Le Lann, and Ulrich Schmid. Failure detection with booting in partially synchronous systems. In *Proceedings of the 5th European Dependable Computing Conference (EDCC-5)*, volume 3463 of *LNCS*, pages 20–37, Budapest, Hungary, April 2005. Springer Verlag.

[4] J.-F. Hermant and Josef Widder. Implementing time free designs for distributed real-time systems (a case study). Research Report 23/2004, Technische Universität Wien, Institut für Technische Informatik, May 2004. Joint Research Report with INRIA Rocquencourt. (Submitted for publication).

[5] Josef Widder. *Distributed Computing in the Presence of Bounded Asynchrony*. PhD thesis, Vienna University of Technology, Fakultät für Informatik, May 2004.

[6] T. K. Srikanth and Sam Toueg. Optimal clock synchronization. *Journal of the ACM*, 34(3):626–645, July 1987.

[7] Josef Widder. Booting clock synchronization in partially synchronous systems. In *Proceedings of the 17th International Symposium on Distributed Computing (DISC'03)*, volume 2848 of *LNCS*, pages 121–135, Sorrento, Italy, October 2003. Springer Verlag.

[8] High Resolution Timers homepage.
`http://high-res-timers.sourceforge.net/`, last tested in April 2005.

[9] The linux kernel preemption project.
`http://kpreempt.sourceforge.net/`, last tested in April 2005.

[10] J.-F. Hermant and Gérard Le Lann. Fast asynchronous uniform consensus in real-time distributed systems. *IEEE Transactions on Computers*, 51(8):931–944, August 2002.

[11] Daniel Albeseder. Experimentelle Verifikation von Synchronitätsannahmen für Computernetzwerke. Diplomarbeit, Embedded Computing Systems Group, Technische Universität Wien, May 2004. (in German).

[12] Daniel Albeseder. Evaluation framework for partial synchronous algorithms in the Θ-model. `http://evalpsa.sourceforge.net`, 2004.

[13] David L. Mills. RFC 1305: Network time protocol (version 3) specification, implementation, March 1992.

[14] Roland Höller, Martin Horauer, Günther Gridling, Nikolaus Kerö, Ulrich Schmid, and Klaus Schossmaier. SynUTC - high precision time synchronization over Ethernet networks. In *Proceedings of the 8th Workshop on Electronics for LHC Experiments (LECC'02)*, pages 428–432, Colmar, France, September 9–13, 2002.

[15] Gérard Le Lann and Ulrich Schmid. How to maximize computing systems coverage. Technical Report 183/1-128, Department of Automation, Technische Universität Wien, April 2003.