The SimUTC Fault-Tolerant Distributed Systems Simulation Toolkit

Bettina Weiss, Günther Gridling, Ulrich Schmid, Klaus Schossmaier
Technische Universität Wien
Department of Automation
Treitlstraße 1, A-1040 Vienna, Austria
{bw, gg, s, kmschoss}@auto.tuwien.ac.at

Abstract

We introduce our SimUTC toolkit, a fault-tolerant distributed systems simulation built upon the discrete event simulation package C++SIM *[2]. SimUTC has been developed in the course of our project SynUTC and targets distributed algorithms for high-accuracy fault-tolerant clock synchronization. This application domain requires detailed simulation models for network transmission and local clock devices, fault-injection capabilities, flexible system configuration facilities, and customized data capture and analysis tools. We explain how SimUTC addresses those issues and provide a few samples of simulation results gathered from the evaluation of the well-known* Fault-Tolerant Average *clock synchronization algorithm.*

Keywords: *discrete event simulation, clock synchronization, fault-tolerant distributed real-time systems, performance evaluation, fault injection.*

1 Motivation

Dealing with distributed algorithms for fault-tolerant real-time systems usually requires more than one technique for validating a solution. In our research project SynUTC[1], for example, we employ analytical methods for deriving correctness proofs and worst-case bounds for precision and accuracy of interval-based clock synchronization algorithms [7]. Those results, however, do not answer questions about the average case behavior — apart from the observation that the worst-case is only attained in quite "exotic" execution runs. Nevertheless, there are applications like fault-location in power distribution grids that primarily require a clock synchronization service with good average accuracy.

Since in-depth analytical modeling of average case performance is difficult, experimental evaluation or simulation must be used for assessment here. Obviously, experimental evaluation has the advantage of inherently providing realistic results, although it has a number of well-known drawbacks:

- The required hardware and software platform has to be at hand.
- Gathering representative data can take much time in real-time execution.
- There is only limited control of the behavior of system components.

Simulation helps to circumvent those problems, at the cost of setting up a suitable tool. First of all, simulation studies can begin as soon as the algorithms are available. Second, the exploitation of simulation time instead of real-time considerably shortens the evaluation process. This is particularly true for any round-based algorithm, where short phases of activity are separated by long (10 ... 100 seconds) idling periods. Last but not least, simulation offers complete control over the behavior of the system with respect to parameters and fault occurences.

Our work on SynUTC thus necessitated an elaborate simulation system for interval-based clock synchronization algorithms in networked distributed systems. Scrutinizing the existing literature, however, we soon recognized that the envisioned SimUTC toolkit had to be built up from the scratch. In fact, none of the existing clock synchronization simulations had been designed for openness and flexibility, but rather targeted to a specific setup. Moreover, neither the simulation framework of [4] nor the one of [1] provides simulation models for network and clocks that

[1]The SynUTC-project received support from the Austrian Science Foundation (FWF) grant P10244-ÖMA, the OeNB "Jubiläumsfonds-Projekt" 6454, the BMWV research contract Zl.601.577/2-IV/B/9/96, and the Austrian START programme Y41-MAT. For further reference see *http://www.auto.tuwien.ac.at/Projects/SynUTC/*

are elaborate enough for high-accuracy clock synchronization. Another major drawback of existing systems is that they do not properly address the issue of long-term simulation, which is mandatory for studying average case behavior. This is particularly true for the simulation system described in [5], which also targets high-accuracy clock synchronization. Since it lacks features like probabilistic fault generation, it is rather a parametrizeable virtual execution environment that supports (basically deterministic) execution of pre-configured scenarios than a fully-fledged simulation environment.

The SimUTC toolkit surveyed in this paper provides a flexible environment for both simulation and experimental evaluation of round-based clock synchronization algorithms in distributed systems. Our description is organized as follows: Section 2 introduces the basic system model and the target hardware. Section 3 provides the requirements specification of the simulated components, Section 4 describes the current implementation. An outline of the evaluation system EvalSys, a GUI-based tool to setup and control SimUTC, is given in Section 5. The description of SimUTC's architecture is completed by an example of practical usage in Section 6 and a discussion of the inherent limitations in Section 7.

2 System Model and Target Architecture

Our primary problem is to synchronize an ensemble of physical clocks that are spatially distributed. Each clock is part of an independent computing system called a *node*, which is connected to one or more data communication networks. In order to achieve clock synchronization in such a setting, each node periodically broadcasts its own clock value to all other nodes in the course of a *full message exchange* (FME). Any node collects and processes the incoming messages from the other nodes, and, near the end of a *synchronization round*, computes a correction value for its clock via a suitable *convergence function* (CF). Thereafter, at a specific point in time, the clock is adjusted, which launches the next synchronization round as shown in Figure 1.

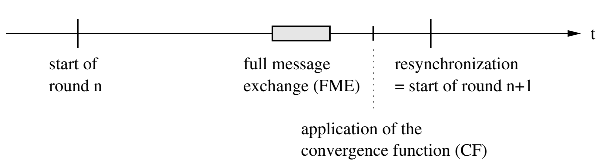

Figure 1. Synchronization Round

The SynUTC-project targets clock synchronization with worst case precision/accuracy in the μs-range in LAN-based system architectures. Following the usual terminology, *precision* stands for the maximum difference between simultaneous clock readings of all pairs of nodes, whereas a node's *accuracy* denotes the difference between its clock value and real-time.

This goal necessitates some hardware support, which is provided by our *Network Time Interface* (NTI) M-Module, see [6] for an overview. Apart from hosting the state- and rate-adjustable clock implemented in our UTCSU-ASIC (*Universal Time Coordinated Clock Synchronization Unit* [9]) and interfacing to the 1 pps output of GPS timing receivers, the NTI is primarily responsible for timestamping all messages that are sent from resp. received at a node. Since timestamping is actually done when a certain byte of the message is put on resp. obtained from the network, neither the varying execution time of our network device driver nor the present network traffic impairs timestamping accuracy. Thus, the atop running clock synchronization algorithm is relieved from all timestamping tasks, but is otherwise not affected by our hardware support.

The target system used for experimental evaluation is a VMEbus-based distributed system consisting of 16 nodes running under ISI's pSOS^{+m} multiprocessor real-time kernel. Apart from CPU and Ethernet network controller, all nodes are equipped with an NTI and execute a custom *clock synchronization application* (CSA). Each CSA consists of the clock synchronization algorithm under study and some monitoring environment that keeps track of its execution. System control and collection of monitoring data is performed by an *evaluation system* (EvalSys) running on a Linux workstation, which communicates with the nodes' CSA via TCP/IP stream sockets. Figure 2 portraits this target system architecture.

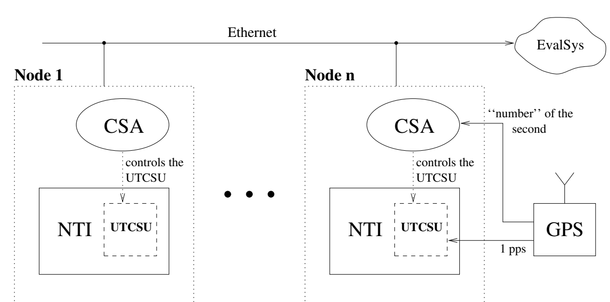

Figure 2. Basic Target System Architecture

When we started to set up the requirements of the SimUTC toolkit, the above hardware was not yet available. Therefore, our idea was to build an evaluation framework that uses *pure simulation* initially, but allows switching to *hardware-based simulation*, i.e., experimental evaluation enriched by system control capabilities, when the hardware is eventually at hand. Therefore, our principal design ensures that the simulated clock can be replaced by a suitable hardware clock without any changes to the modules in the CSA. Similarly, the simulated network can be replaced by a real network + NTIs without any changes. This is achieved by well-defined interfaces to clock and network, which must of course be implemented prior to integrating a particular hardware component. Moreover, both simulated clock and network must be replaced by their real counterparts simultaneously. We should point out, though, that not all features offered in pure simulation are also available in hardware-based simulation. There are, however, enough ways to affect the behavior of the real clock and network to warrant the term hardware-based simulation instead of sole experimental evaluation.

3 Requirements Specification of the Simulation System

During the requirements analysis of clock synchronization in a target system like the one of Section 2, we identified four major components and their parameters of interest. They are outlined in the following subsections.

3.1 Network

The purpose of this component is to transmit messages to remote nodes and to receive/announce incoming messages. The key parameters are the *transmission delay characteristics* of the network, which consist of the deterministic (δ_{pq}) and maximum indeterministic ($\epsilon^{\pm}_{pq}$) part of the transmission delay between each pair of nodes p, q along with a distribution function defined on the range $[\delta_{pq} - \epsilon^{-}_{pq}, \delta_{pq} + \epsilon^{+}_{pq}]$. Apart from predefined functions `Uniform`, `Exponential`, `Normal`, `Gamma`, `Weibull`, a user-defined function table can be supplied as well.

In addition to this regular transmission behavior, one can also set upper and lower bounds exceeding $[\delta_{pq} - \epsilon^{-}_{pq}, \delta_{pq} + \epsilon^{+}_{pq}]$. They are used during fault injection, where messages travelling faster or slower than allowed in normal operation may be generated.

Apart from transmission delays, there are two additional delay parameters *broadcast latency* and *broadcast operation delay* that model different network topologies. The former can be used to account for contention on shared channels like Ethernet, the latter applies to non-instantaneous broadcasting as in point-to-point networks.

All network parameters introduced above are available in pure simulation only, although there are some limited means for controlling transmission delays and fault injection in hardware-based simulation as well. For example, although there is no way to make message transmission faster here, it is nevertheless possible to further *delay* outgoing and/or incoming messages on a node. Any message can also be scrambled, which is done according to a user-defineable *probability of scrambling*.

3.2 Clock

This component keeps track of the local time on a node and allows the algorithm to read and also set its state. Setting the clock is possible by means of two parameters that allow access to the clock state, namely, the *clock value* and the *accuracy interval*, where the latter is needed by interval-based synchronization algorithms [7]. Note that those features can be used to establish a certain initial synchronization condition as well.

Clock fault injection is controlled via the probabilities for clock *read* and *write faults*. If such a fault occurs, the read or write access to the clock results in an arbitrary value.

As far as clock drift is concerned, one can specify the constant average drift of any clock via the *initial clock drift* parameter. In pure simulation, this is useful to account for the inevitable (manufacturing-dependent) initial frequency offset of a hardware clock. Of course, a constant drift would be too weak for accurately modeling the actual drift behavior of such devices, hence capabilities for both *random* and *deterministic drift changes* are provided by SimUTC.

Random drift changes model spontaneous drift “jumps” as caused by frequency noise, which may even lead to a drift outside the nominal range in case of faulty clocks. A slow, regular, and fast drift range along with appropriate *probabilities of a drift change* can be specified for each clock. Whenever a clock is accessed, its initial drift may be modified by a value drawn from one of these ranges.

Deterministic drift changes can be used for modeling dynamic changes due to deterministic effects like temperature changes. The user can set parameters that cause SimUTC to sweep a clock’s drift over a certain range within a given period of time, independent of clock accesses.

All those features are available both in pure and in hardware-based simulation.

3.3 Algorithm

The algorithm periodically triggers the activities shown in Figure 1. A number of parameters allow to specify the configuration of the clock synchronization algorithm. Among those are the timing characteristics of the synchronization round: The *round period* specifies the duration of the full round, the *delay compensation* gives the time from the start of the FME until the application of the convergence function, and the *computation delay compensation* denotes the time from the application of the CF to resynchronization.

Other parameters required for most clock synchronization algorithms are the *number of nodes* and the *number of faulty nodes*. The former gives the total number of nodes, whereas the latter denotes the maximum number of faulty nodes that the algorithm can tolerate.

An important issue for high-accuracy clock synchronization is clock granularity, which can be chosen via the *clock timestamp granularity*.

3.4 Simulation control

This component is designed to modify and monitor the simulation environment transparently to the algorithm under study. The algorithm used in a particular evaluation can be chosen interactively from the list of implemented ones. The current version of SimUTC comes with a few built-in examples like FTA (see Section 6), which can easily be extended by additional algorithms written in C++ atop of a pre-defined interface to clock and network, see [8] for an example.

In order to be able to temporarily take out a node, the user can *suspend* or *resume* a single CSA or all of them. A suspended CSA stops working and lets its clock drift until it is resumed again.

Two commands enable or disable system monitoring via *snapshots*. When snapshots are enabled, all CSAs periodically read their clock values at the same time and send this value to the evaluation system. These values are used by EvalSys to monitor the overall system's precision and accuracy, see Section 5. The period of the snapshots can be configured.

4 Architecture of the Simulation System

In order to implement the components sketched above, we mapped them onto a few *modules* with well-defined functions and interfaces. Figure 3 shows a more detailed view of the SimUTC architecture, incorporating the simulation controller, a network module, two CSAs "running" on nodes p and q, and dashed control paths between the modules. Note that some components are coded within several modules, like the network, which actually comprises `Lance` and `Network`.

The `Controller` is the main module that commands the whole simulation. After starting SimUTC, only the `Controller` exists and awaits messages from the evaluation system. The user can then create CSAs, configure the system, and finally start the simulation. Each CSA involves four modules:

- The `Supervisor` module is like a `Controller` and has its own link to EvalSys, but is only responsible for its CSA. It configures it according to the user's demands, periodically takes a snapshot of its `Clock`, and reports its monitoring data back to EvalSys.

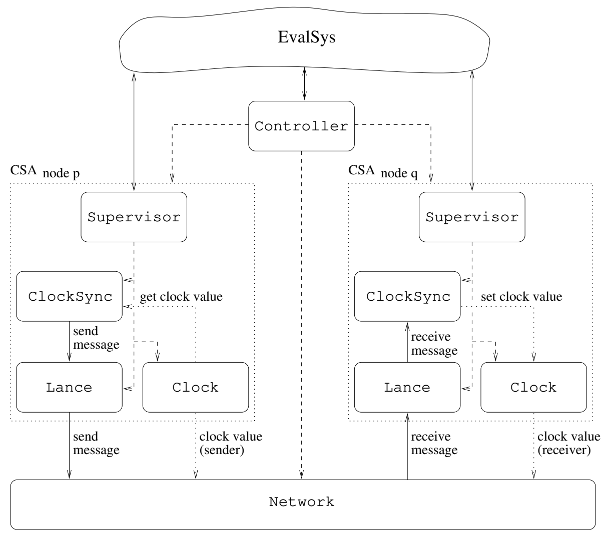

Figure 3. SimUTC Software Architecture

- The `ClockSync` module embeds the actual clock synchronization algorithm. Basically, it offers an environment that enforces the round structure depicted in Figure 1 and mediates access to the `Lance` and `Clock` modules. More specifically, `ClockSync` distributes the local clock data, collects messages from remote nodes and passes their data to the clock synchronization algorithm, calls the CF at the appropriate time, and finally requests clock correction at the end of the round. The user-supplied clock synchronization algorithm itself is simply a C++ class with a well-defined interface, which provides functions for storing the clock state and rate data of the remote nodes, the particular convergence function, and the method of clock correction. Hence, it provides the algorithm-specific functions, whereas `ClockSync` decides when to call them.
- The `Clock` module allows access to the clock of the node. It comprises the clock simulation itself, and an interface class to it. Given our project's focus towards interval-based clock synchronization in conjunction with our UTCSU-ASIC, its interface is closely modeled on the latter and thus includes non-standard functionality. More traditional clock synchronization algorithms typically rely upon a subset of the interface and use only the functions for reading and setting the clock.
- The `Lance` module is used as an interface to the network. This module is very simple; its main use lies in delaying and scrambling messages during fault injection.

In order to facilitate message transmission in the pure simulation setup, a `Network` module is needed. It is also under the supervision of the `Controller` module. According to Subsection 3.1, SimUTC supports point-to-point and broadcast network topologies. Note that our current broadcast network implementation differs from a typical one like Ethernet in that it allows the reception order of messages sent by a specific node to be different from the sending order. Of course, SimUTC can be extended by additional network components, which can then be selected via EvalSys at runtime, similar to selecting the algorithm type.

Figure 3 also shows the dependencies between those modules of a CSA that actually take part in clock synchronization: In the course of a full message exchange, `ClockSync` hands over a broadcast message to the `Lance`, which may delay and scramble the message during fault injection and finally passes it on to the `Network`. Here it is delayed to simulate a *broadcast delay* (e.g., caused by medium access conflicts), timestamped on transmission with the local time of the sender node (as is done in hardware by the NTI), delayed again for some time to simulate the *transmission delay*, and finally timestamped with the local time at the respective receiving node. The message is then announced to the `Lance` module, which in turn passes it on to its `ClockSync` module, possibly delaying and scrambling it during fault injection. At the end of the synchronization round, the `ClockSync` module uses all information collected during that round to compute a correction value and sets the clock accordingly.

In pure simulation, all events in the above modules are executed as separate processes under the discrete-event simulation package C++SIM from [2] running on a single Linux-workstation. C++SIM offers C++ classes for processes, queues, events, etc. and is based on threads, which are co-ordinated by a built-in scheduler. The scheduling strategy is non-preemptive, so threads —and therefore processes— are allowed to execute until they suspend themselves. To simulate the progress of time, C++SIM keeps its own *simulation time* t_{sim}, a fictive value that is used to determine when processes become active. At the start of the simulation, t_{sim} is set to zero. The C++SIM-scheduler stores all processes in a list sorted by their activation times $t_p \geq t_{sim}$ and always executes the first process in the list. Before a process is actually started, however, t_{sim} is advanced to its activation time t_p and stays at this value until the running process suspends itself, i.e., until the next process is about to be scheduled.

In SimUTC, simulation time models real-time. C++SIM processes are used to schedule events, like clock interrupts or message arrivals, at the times they would occur in the real hardware. Note that this ensures that the results of pure and hardware-based simulation will be comparable. In response to an event, SimUTC computes the system state valid at this simulation time, executes the code that should react to the event (and whose actions result in the scheduling of new events), and finally suspends the process when it is finished. Therefore, the time between events passes instantaneously and the system state is only computed when necessary. The drawback of our design is that, from the point of view of the simulation, the execution of processes happens instantaneously; the execution times of functions are not implicitly taken into account. Nevertheless, we opted for the faster execution which is a valuable feature for long-term evaluation.

When switching to hardware-based simulation, the whole C++SIM package can be removed. The simulation is now executed concurrently on multiple nodes, the local scheduler is provided by each node's pSOS^{+m} kernel. The `Supervisor` and `Controller` modules are pSOS^{+m} tasks, events are now triggered by hardware interrupts from the network and the clock. The `Network` module is replaced by the hardware and its pSOS^{+m} device-driver counterpart, the `Clock` module is reduced to the clock interface class, which accesses the registers of the real clock and contains the code for influencing the clock's behavior. To the clock synchronization algorithm, however, they provide the same interface as the pure simulation modules.

Despite the fact that hardware-based simulation uses real components, it still retains most of its system control capabilities: Even though a node's local clock is now driven by a real oscillator with unknown drift, the adder-based clock in the UTCSU allows all types of drift modification mentioned in Section 3. The only severe loss of control concerns the network, which can only introduce additional delays in outgoing or incoming messages (through `Lance`), but cannot influence the transmission delay on the network itself. Of course, as in pure simulation, `Lance` can also generate packet losses or message scrambling.

Whereas it was easy to implement global (simulation) time and simultaneous snapshots on all CSAs in pure simulation, those tasks require special hardware support in hardware-based simulation. Global time is provided by means of an additional NTI per node. Since they are all driven by a common oscillator and simultaneously started by using the UTCSU's *SyncRun*-pin, the "global time" UTCSUs operate in lockstep and, hence, always display the same time at all nodes. Simultaneous snapshots are implemented by means of the snapshot feature of the UTCSU, which facilitates automatic timestamping of external events. The *Snapshot*-pins of all UTCSUs in the system are tied together and driven by a programmable output pin of the NTI at a particular node. This node can simultaneously trigger snapshots at all clocks in the system.

5 Evaluation System

The X-windows based graphical user interface EvalSys allows the user to configure and command SimUTC and to carry out data analysis. Simulation control is achieved by sending command and parameter messages over dedicated stream sockets. Similarily, event messages reported by the simulation on these sockets are used to gather monitoring data from all CSAs.

After starting the program, EvalSys first builds the simulation environment by creating nodes and network and by configuring all parameters as desired. Once the environment is established, the simulation starts sending event reports to notify the evaluation about the system state. These events are collected by EvalSys, which extracts precision and accuracy data and displays them on the screen. At any time during the session, the user can change parameter settings to modify the environment. The simulation continues to work until it is explicitly stopped by the user. Note that EvalSys keeps all data it has collected, even after the simulation has been terminated, and allows to save it for further reference in later sessions.

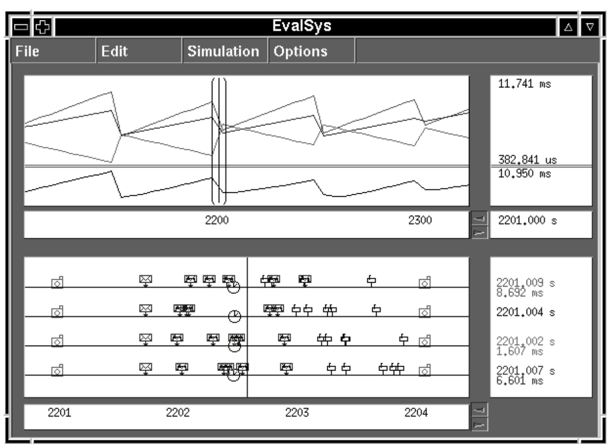

Figure 4. EvalSys Screenshot

Figure 4 shows the user interface of EvalSys. The lower half of the application is consumed by the *event stream window*, which displays a user-definable selection of the SimUTC events reported by each node. SimUTC offers several event types, each of which contains specific information about the system state. The most important event is the `Snapshot`, which consists of the simulation time and the instantaneous local clock value. All nodes generate this event periodically, at the same simulation time, so it can be used to compute precision and accuracy of all nodes in the system. The events are sorted by simulation time, which serves as a global time. Clicking on an event icon shows the data of this event. Snapshot data at a certain time can be viewed by setting a *position marker*, which is represented by the vertical line.

The *precision window* situated above the event stream shows the global (= maximum) precision in its lower part, and the accuracy of each node participating in the synchronization in its upper part. The accuracy graphs can be selectively hidden to avoid a cluttered display. Two curved vertical lines represent the part of the whole stream that is displayed in the event stream window. The position selected in the event stream is again marked with a straight vertical line.

The *state window* to the right is parted into a display of global data next to the precision window and a display of node-specific data next to the event stream window. Node-specific data consist of the clock value at the snapshot time and, optionally, its accuracy. The maximum and minimum accuracy encountered for the selected nodes are shown next to the accuracy graphs. Below, the worst case precision of the current simulation run is printed. The small window in the middle displays the simulation time of the last snapshot before the marker position.

At any time, the user can scroll over the event stream and zoom into it to get a more detailed view of interesting sections. Additional means of navigation are available, for example finding local or global maxima or minima in the precision graph. User-definable search conditions help to filter out or to find given system states.

In addition, EvalSys can print certain simulation data into files in gnuplot format [10]. The data is taken from the snapshot events and plotted over the simulation time. This feature allows further processing of the data with a simple text editor and to incorporate it into documents. In Section 6, we will provide a few examples:

- *Accuracy* plots the deviation of a node's clock value from the simulation time at which this value was read.
- *Precision* is the maximum difference of simultaneous clock readings, taken over all pairs of nodes.
- *Adjustment* shows the difference between the old clock value and the new clock value enforced by the synchronization algorithm.

A major requirement put on the development of EvalSys was maximum flexibility with respect to modifications of the simulation. Consequently, the layout of all message data structures used in the communication between EvalSys and SimUTC is loaded into EvalSys from text files at startup, which makes it easy to integrate new algorithms or networks into the simulation. System configuration itself can be automated via parametrized macros and saved for future sessions.

6 Example

In this section, we round off our overview of SimUTC's features by means of a simulation run for the well-know *Fault-Tolerant Average* (FTA) clock synchronization algorithm [3]. The simulation model for our experiment is as follows:

- The ensemble consists of 5 non-faulty nodes without access to a GPS-receiver.
- The respective average clock drifts are 0 ppm, 0.5 ppm, -0.5 ppm, -1 ppm, and 2 ppm. The continuous drift changes are within ± 1.5 ppm.
- The nodes are attached to a non-faulty broadcast network with a maximum latency of 1 ms.
- The transmission delay obeys a Normal distribution in the range of 50 μs to 80 μs with an expectation of 60 μs.
- The round period is set to 12 s and the delay compensations amount to 1.8 ms.
- The snapshot period is 3 s and the simulation terminates after 300 exchanged messages.

Figure 5 shows three plots gathered from the simulation run for the FTA function. The top one shows the accuracy of each clock, which clearly illustrates the periodic resynchronizations and drift phases in between. Moreover, we can verify the individual clock drifts, whereas the whole ensemble drifts away from the simulation time by approximately 0.4 ppm.

The middle plot in Figure 5 shows the precision, which is mainly determined by the slowest node 4 and the fastest node 5. It has the expected sawtooth-like shape, where the endpoints are shifted according to the encountered transmission delay uncertainty. In this particular run, the precision wanders roughly between 5 μs and 42 μs. Of course, to actually evaluate the performance of synchronization algorithms, statistical data from sufficiently many runs must be collected.

Finally, the bottom plot in Figure 5 exhibits the adjustment of each clock at the resynchronization time. For instance, node 3 experiences an advancement of about 10 μs at every resynchronization instant. Note that all nodes measure the average round-trip delays of their messages before embarking with the synchronization, with nodes 1 and 5 starting early.

7 Limitations

The most obvious problem to the user is the lack of simulation control related to some specific point in simulation time: Although the user can change parameters during runtime, it is uncertain when the simulation will react to them.

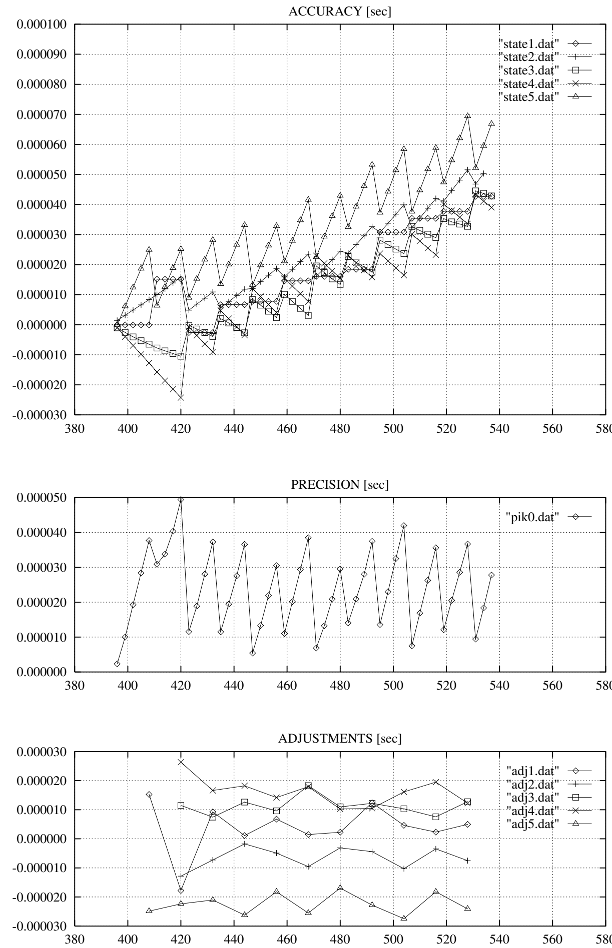

Figure 5. Simulation Run using FTA

The problem is particularly annoying in pure simulation, where the time between events passes "instantaneously". In hardware-based simulation, the situation is more relaxed, since the non-zero real-time between events is available for setting parameters or issuing commands. Nevertheless, even in hardware-based simulation there are no guarantees that a modification will be in effect soon enough.

Another difficulty related to this lack of coordination is the fact that EvalSys has no means to order events from different nodes, each of which has its own local time and its own socket to send the events. Thus, the nodes themselves must provide enough information not only to order events but even to place them on a common time scale. This leads to the requirement of a global time maintained by the simulation and provided to each node, which is easy to achieve in pure simulation, but requires the use of dedicated and externally synchronized clocks in hardware-based simulation. This calls for additional hardware support, which is implemented by means of dedicated NTIs in our setting.

In pure simulation, our discrete simulation approach implies that all events are handled instantaneously. We do not consider delays which result from the execution time of the code. This greatly simplifies the simulation, but problems with overly long functions or with race conditions cannot be found. They will only occur in hardware-based simulation.

Designing the pure simulation system for maximum compatibility with the hardware-based one faces the trouble that we cannot exploit any global knowledge the pure simulation running on a single workstation might have. More specifically, we are forced to assume that the nodes are physically separated and do not have information on the state of their peers. Hence, we cannot enforce a global fault model such as the usual one that no more than $f = \lfloor (n-1)/3 \rfloor$ nodes suffer from a byzantine fault. So it is up to the user to either configure the fault occurence probabilities conservatively enough to make its behavior comply to the fault model, or to accept that sometimes the simulation will transgress the model. Although this severely limits the usefulness of SimUTC for detailed simulation studies of fault behavior, it is nevertheless the right model for average case performance studies – even the convenient global fault models must eventually be mapped to real systems.

Last but not least, at this time we do not have any results collected with hardware-based simulation, which would help to verify the data obtained with pure simulation. Although the results look sound, unnoticed implementation errors could still invalidate the data. We are currently working on setting up the hardware evaluation environment, so hardware-based simulation results should be available soon.

Conclusions

We presented an overview of our discrete event-simulation toolkit SimUTC designed for the evaluation of clock synchronization algorithms in networked distributed systems. Due to its layered, modular and distributed architecture, SimUTC is a flexible and powerful instrument for evaluating the performance of such algorithms.

SimUTC provides elaborate simulation models for both network and clocks. In fact, when aiming at high accuracy clock synchronization in the 1 μs range, many system parameters ranging from clock granularities up to oscillator stability must be taken into account and, hence, appropriately modeled. Various fault-injection capabilities in conjunction with customized data analysis features facilitate in-depth simulation studies over long periods. All SimUTC functions are controlled via a user-friendly GUI, which supports system configuration, simulation control, and data analysis.

A key issue in the design of SimUTC has been the full compatibility with hardware-based simulation. Therefore, it is possible to replace the simulated network and clock modules by real network controllers and clock devices with minimal changes. A forthcoming paper will be devoted to this framework for comprehensive experimental evaluation.

References

[1] C. Fetzer and F. Cristian. Building fault-tolerant hardware clocks. In *Proceedings of the Seventh IFIP International Working Conference on Dependable Computing for Critical Applications*, pages 59–78, San Jose, USA, January 1999. http://www.cs.ucsd.edu/˜cfetzer/HWC.

[2] M. C. Little and D. McCue. Construction and use of a simulation package in C++. *C User's Journal*, 12(3), 1994. Postscript file available at http://cxxsim.ncl.ac.uk/homepage.html.

[3] J. Lundelius-Welch and N. A. Lynch. A new fault-tolerant algorithm for clock synchronization. *Information and Computation*, 77(1):1–36, 1988.

[4] M. J. Pfluegl and D. M. Blough. Evaluation of a new algorithm for fault-tolerant clock synchronization. In *Proc. Pacific Rim Internation Symposium on Fault Tolerant Systems*, pages 38–43, Kawasaki, Japan, September 1991.

[5] A. Schedl. *Design and Simulation of Clock Synchronization in Distributed Systems*. PhD thesis, Faculty of Technical and Natural Sciences, Vienna University of Technology, April 1996.

[6] U. Schmid, J. Klasek, T. Mandl, H. Nachtnebel, G. R. Cadek, and N. Kerö. A Network Time Interface M-Module for distributing GPS-time over LANs. *J. Real-Time Systems*, 17(2), 1999.

[7] U. Schmid and K. Schossmaier. Interval-based clock synchronization. *J. Real-Time Systems*, 12(2):173–228, March 1997.

[8] K. Schossmaier and J. Klasek. Implementing the optimal precision algorithm for clock state & rate synchronization. Technical Report 183/1-91, Department of Automation, TU Vienna, February 1999.

[9] K. Schossmaier, U. Schmid, M. Horauer, and D. Loy. Specification and implementation of the Universal Time Coordinated Synchronization Unit (UTCSU). *J. Real-Time Systems*, 12(3):295–327, February 1997.

[10] T. Williams and C. Kelley. GNUPLOT – An interactive plotting program. Program available e.g. at ftp://ftp.dartmouth.edu/pub/gnuplot/, 1993.