SimUTC
User Manual

project SimUTC

version 2.3.1.0

B. Weiss

February 8, 1999

Abstract

SimUTC is a simulation program for evaluating distributed interval-based clock synchronization algorithms. The program, which is written in C++, uses a discrete-event approach to simulate the environment of such algorithms, and allows the user to control this environment, thus exposing the clock synchronization algorithm to well-known conditions. The behaviour of the algorithm, which is represented by the value of the clock that it controls, is monitored and made available to the user.

SimUTC has been developed for execution on a PC using Linux as operating system. The simulation package is freely available and includes the source code, the manuals in postscript format, and HTML descriptions of the source files. See the last page of this manual on how to get and extract the archive. SimUTC requires the freely available simulation tool C++SIM [LM] and the REX lightweight process library [REX].

The following text can be seen as an extended User Manual. It explains the motivation for developing such a simulation, introduces a few necessary terms, and then describes the user interface of the simulation program.

Contents

List of Figures

1 Introduction

In order to synchronize clocks in a distributed system, the nodes in the system have to exchange information about their local clock values (typically by periodically sending their own clock value over a network to all other nodes) and to process the data from the other nodes to compute new clock values for their own local clocks. The algorithm that is used for the synchronization should be fault-tolerant – at least up to a certain degree – and its behaviour in case of faults should be fully understood. But since faults do not occur very often in todays computer systems, and even if they happen they cannot be traced back most of the time, it is not possible to evaluate an algorithm thoroughly and systematically by just executing it on a target system. Therefore a simulation environment is needed that can inject faults into the correct behaviour of the underlying hardware. Moreover, since testing on the actual hardware is not always possible, the simulation environment should in fact provide the possibility to simulate the hardware entirely (node, network). For the user, who controls the simulation, injects faults, and observes the system's behaviour, it should be as transparent as possible whether the clock synchronization algorithm is being executed on the real hardware or just in the simulation environment. Simulating the hardware will offer a better control over the system behaviour, but apart from that the user should not note any difference. This full control over the environment does also facilitate comparisons between different clock synchronization algorithms under exactly the same conditions, which enables the user to "fine-tune" an algorithm for a certain target environment.

Our project SynUTC[1] [SynUTC] is devoted to the development of soft- and hardware for very high precision/accuracy clock synchronization in fault-tolerant distributed systems. More specifically, by executing interval-based clock synchronization algorithms [SS97], [Scho97] on the Network Time Interface (NTI) hardware [HSS98] built around our novel UTCSU (Universal Time Coordinated Synchronization Unit)-ASIC [Loy96], any node in a LAN-based distributed system can display a reliable local time with a precision and accuracy (w.r.t. UTC) in the 1μs range. Whereas testing the hardware is a relatively straightforward process, the synchronization algorithm has to be fault-tolerant with respect to a wide variety of faults, hence a simulation environment is needed to simulate them.

Our analysis of the problem revealed that there are three elements that a simulation should include:

1. It should simulate the hardware and monitor the behaviour of the synchronization algorithm under normal (fault-free) conditions.

[1] A co-operation of the Department of Automation and the Department of Computer Technology, both Vienna University of Technology, A-1040 Vienna, Austria.

2. It should allow the user to control the occurrence of failures in the hardware.

3. It should enable the user to repeat simulation runs with slightly modified conditions.

The first feature, and in part also the second one, are needed for a long-term evaluation of an algorithm under given conditions, whereas the third feature comes in handy if some interesting behaviour of the algorithm has to be examined systematically under slightly different conditions.

This manual is concerned with parts (1) and (2), i.e., developing a basic simulation environment that can simulate the whole hardware (the default, since the hardware is not yet available) and – with slight modifications – be executed on real target nodes. The simulation also offers means to specify faulty behaviour of a node. Development of the simulation was centered on two requirements: making the switch from simulated hardware to the real one as easy as possible, and designing the system as flexible as possible with respect to exchanging the clock synchronization algorithm employed.

In Section 2, I will first explain a few terms used in the rest of the text and give a short explanation of the principles of clock synchronization as it is implemented in the simulation. Section 3 describes the main features of the target hardware and outlines the structure of the simulation system, which is broken down into several modules. Section 4 gives a more detailed description of each module. Section 5 will explain how to integrate user-supplied synchronization algorithms, and Section 6 shows how to add new network types to the simulation, which is very similar to adding a new algorithm.

An own evaluation system is needed for controlling the simulation and visualizing its data, so its programmer (and maybe also the user) will especially be interested in the Evaluation Interface Manual, which is dealing with the user (i.e., evaluation) interface of the simulation.

The Hardware Portation Manual contains instructions on porting the simulation software to the hardware available in SynUTC.

2 Clock Synchronization

Our problem is to synchronize an ensemble of clocks that are spatially distributed. Each of these clocks is part of an independent computing system which is called a node. The nodes are connected to one or more networks, and thus can send their clock values to other nodes.

Clock synchronization comes in two flavours:

- *Internal synchronization*, which means that all clocks strive to be as close together as possible, but do not bear any relationship to some global time outside the ensemble.

- *External synchronization*, which means that the clocks try to approach a global reference time existing outside the ensemble (like UTC).

The measure of the quality of internal synchronization is *precision*, which denotes the worst case difference of any simultaneous clock reading over all pairs of clocks. The corresponding term for external synchronization is *accuracy*, the worst case deviation from the reference time over all clocks.

Of course, in order to synchronize to reference time, some nodes need to have access to this time. In SynUTC, up to three GPS receivers can be connected to a node. A node equipped with such a receiver is called a *P(rimary)-Node*, whereas nodes without direct access to the external time are called *S(econdary)-Nodes*.

Nodes may be connected to more than one network, thus the whole system is split into several sets of nodes sharing a network, forming sub-networks. Most nodes in such a subnet do not have access to more than one network; however, the nodes which are indeed connected to more than one network are called *G-Nodes*. They introduce a whole range of new problems to the synchronization, since they are confronted with the data from all their subnets, which may be inconsistent.

At this time, we do not have a solution for the treatment of G-Nodes. Therefore, I have made the assumption that they only take fully part in the synchronization of one of their networks. The subnet in which a G-Node participates is called the *associated network*. Only the data received from this network is used as input for the synchronization algorithm. Of course, the associated network is not fixed and may change in the course of synchronization.

In order to achieve the synchronization of clocks in such a topology, each node periodically has to broadcast its own clock value to the other nodes in the course of a *full message exchange* (FME). The basic period is called the *synchronization round (period)*. The node has to collect and process the messages from the other nodes, and, near the end of the round, compute a new clock value via a *convergence function* (CV). Thereafter, at a specific point in time, the clock is set to the new value, launching the next round. Fig. 1 shows such a synchronization round.

Of course, all nodes should exchange their clock values and set their clocks at approximately the same time. This is not difficult if the clocks are already synchronized, but may pose problems during startup of the system or if a new node is added to an active system.

Processing the data obtained from the other nodes poses additional problems. If a node p receives a message containing the clock value of node q, it cannot immediately use that value. After all, the clock value was drawn when q sent the message, so the *transmission delay* of the message (the time it took the message to travel from q to p) has to be added. Note that the delay depends on the network type, hence it cannot always be known a priori and must be (re-)measured online.

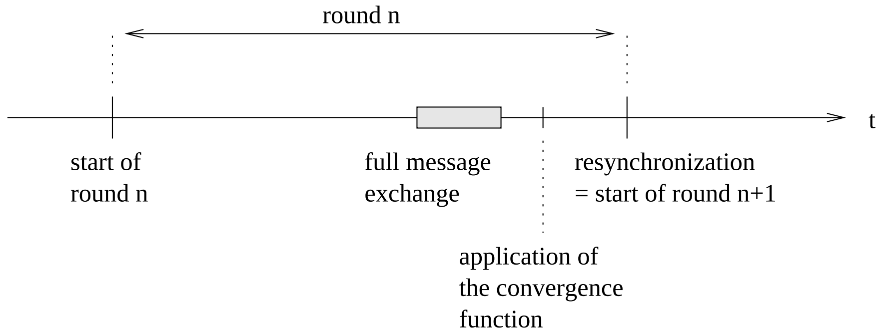

Figure 1: A Synchronization Period

Knowing the transmission delay is not enough, since the remote clock will have a certain clock drift which must be considered as well if p has to accurately compute the clock value of q at the time of message reception. Hence, information about the clock drifts is needed, either a priori or by virtue of rate synchronization (see [Scho97]).

This introduction is sufficient to understand the following description of the simulation system. See [HSS98] for an excellent overview on interval-based clock synchronization. For a more detailed explanation of the synchronization algorithm, see [SS97].

3 System Overview

I use the term *hardware-based simulation* for the experimental evaluation of clock synchronization algorithms, which is our final goal. Since the hardware environment is not yet available, the thesis concentrates on the *pure simulation*, which additionally simulates the behaviour of all hardware components. Nevertheless, the pure simulation was developed with a certain target hardware in mind, presented in the next section.

3.1 Hardware-based Simulation

Fig. 2 represents the hardware layout of the real target system used for the experimental evaluation. We have $m \leq 4$ nodes with a CPU and the operating system pSOS^{+m}, a multi-processing, multi-tasking OS. Each node contains up to 4 *areas* sharing its CPU[2]. In consequence, in our system clock synchronization applications are identified by a pair $\langle node,\ area \rangle$, whereas normally node and area are the same. The nodes are connected to each other via a (broadcast)

[2]The reason for this somewhat awkward configuration is simply reduction of hardware (CPU boards) and therefore of cost.

network. Note that if a clock synchronization message is received on a node, it is provided to all areas on that node.

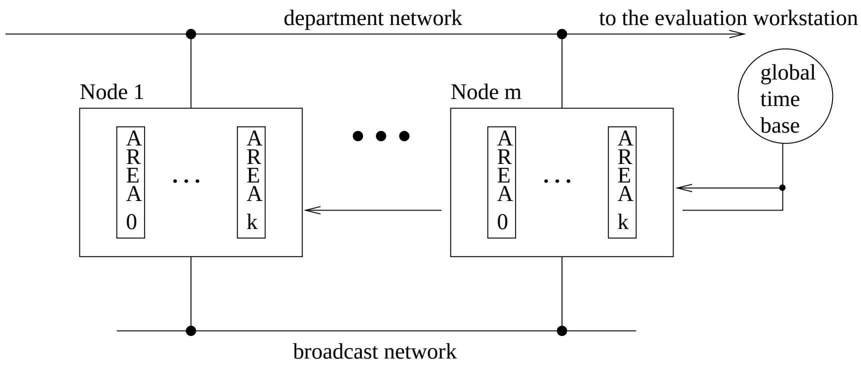

Figure 2: Target System

Since the target nodes are also connected to the network of the department (Ethernet), an evaluation workstation (a PC with Linux and X-Windows) will be used for controlling the whole experimental evaluation and visualizing its data. The simulation uses TCP/IP stream sockets to send its simulation data to the evaluation program. Moreover, the evaluation uses the same sockets to pass user commands to the simulation, thus controlling the execution of the program.

The simulation program, replicated on every node, will report *events* containing its data to the evaluation. For event-ordering purposes, a *global time base* available to all nodes is required. The time base needs not be accurate w.r.t. external time, but obtaining its value should be fast.

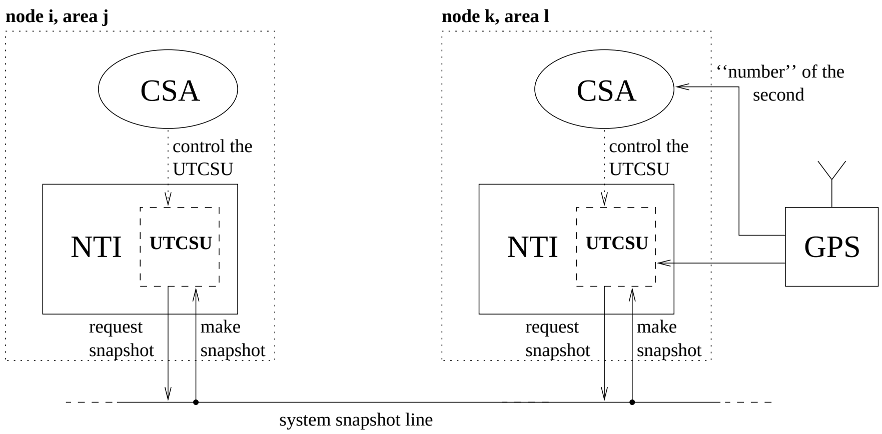

Figure 3: Target System Areas

Fig. 3 outlines the components of an area that are important for the simulation. Each area contains an NTI (Network Time Interface), which is responsible for timestamping all messages that are sent from resp. received at a node *at the exact time* a certain byte is put on resp. obtained from the network. The advantage is that the execution times of the application program and of the network driver functions do not have to be considered, neither does high network traffic invalidate the timestamp. The NTI hosts our physical clock device (UTCSU), from which the clock value for a message timestamp is drawn. Each UTCSU is associated with a *clock synchronization application* (CSA) that is solely dedicated to the clock of that area.

The UTCSU itself contains an adder-based clock providing a 56 bit available time value with a resolution of 2^{-24} seconds. The 56 bits are hold in two 32 bit registers, namely *macrostamp* containing bits [+31,+8], and *timestamp* holding bits [+7,-24]. The numbers of the bits are chosen so that bit 0 denotes 1 second, and a binary format is used to represent the time value.

To support an interval-based approach to synchronization, the clock chip keeps track of the current accuracy of the clock (i.e., the worst case deviation of its clock time value from the reference time), which is modelled as two accuracies (an upper α^+ and a lower α^- one) around the clock time. Both accuracies can be read at once. When reading the accuracies, they correspond to the bits [-8,-23]. The setting granularity is much higher, there you can set bits S:[-8,-38] of the accuracies (only both at once, and to the same value), where S denotes a sign bit.

The macrostamp, timestamp, and accuracies form the *clock value* that is sent to the other nodes. Note that accessing a clock value is atomic. See [Loy96] for an in-depth treatment of the features of the UTCSU, and [HSS98] for a description of the coupling between UTCSU and NTI.

Since the amount of time added to the clock time value at every oscillator tick is adjustable, the clock rate can be influenced by changing this value. Unfortunately, no clock oscillator is perfectly stable, so even if a clock is programmed once to have an ideal rate it might eventually exhibit a non-zero drift. However, big differences between clock rates have an adverse effect on synchronization, so the clock rates should additionally be synchronized (cf. [Scho97]).

In order to evaluate the quality of clock synchronization, we need to take periodic *system snapshots*, that is, sample the clock values of all clocks of the system at simultaneous times. Fortunately, the UTCSU includes a feature to copy its clock value into dedicated registers upon an external event (called a *hardware snapshot*), and has a programmable output pin to create external events. Thus, by wiring all UTCSUs together, every clock can trigger such a snapshot on every other clock, including its own (see Fig. 3). The snapshot data are then sent to the evaluation.

As already mentioned, P-Nodes are connected to a GPS timing receiver. Every second, the receiver produces a 1PPS pulse that indicates the beginning of a current UTC second. This 1PPS pulse is fed into the UTCSU. Apart from the

1PPS signal, such receivers provide further information about the current time (namely the "number" of the UTC second) through a serial interface (RS-232), which is available 10-100 ms after the 1PPS transition. This information is used by the CSA to determine the accurate clock value of the UTCSU.

3.2 Pure Simulation

In this version the simulation system simulates the hardware components of the target system (as far as necessary). The rest of the thesis will be concerned with its presentation.

The operating system used in hardware-based simulation is pSOS^{+m}, which supports multi-tasking, and most activities of the clock synchronization application are triggered by interrupts. In consequence, hardware-based simulation consists of several tasks and makes heavy use of interrupts. As long as the hardware is not available, the simulation is executed in a UNIX environment (Linux). Instead of tasks and interrupts, I use a thread-based simulation package called C++SIM [LM], which facilitates discrete-event simulation. The threads are provided by the REX lightweight process library [REX]. The simulation code is implemented in C++.

Note that the hardware-based simulation must be executed on every target node, so it is replicated up to four times. In pure simulation, however, only one instance of the program is executed, which is responsible for simulating the different nodes (including areas) of the hardware.

In order to simulate the target environment, I have split the program into several modules as exemplified in Fig. 4. It shows the simulation controller, a network, one application "running" on node i, area j, and the control paths between the modules.

The `Controller` is the main module and controls the whole simulation. When you start the program, only the `Controller` will exist and wait for messages from the evaluation system. For communication to `EvalSys`, sockets are used. Whereas the hardware-based simulation has to use Internet sockets, the pure simulation assumes that it is executed on the same workstation as the evaluation system and thus utilizes UNIX sockets.

To simulate message transmissions of the broadcast network, one or more `Network` modules are needed, which are directly under the control of the `Controller` module.

A clock synchronization application (CSA) itself consists of four modules: The `Supervisor` controls the CSA. It is like a `Controller` module and has its own socket to `EvalSys`, but is only responsible for its CSA. Through it, the `Controller` can suspend/resume the CSA, although it has no direct access to its modules. The `ClockSync` module contains the actual clock synchronization

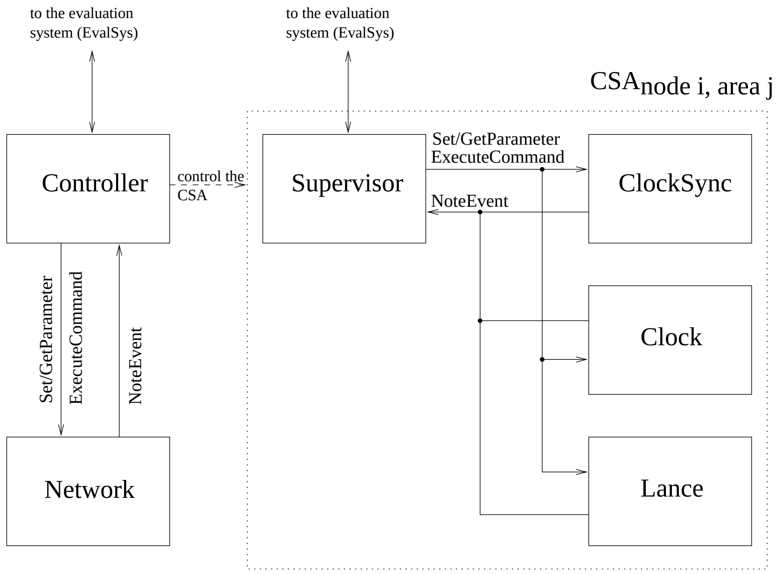

Figure 4: Simulation System Overview

algorithm. Module `Clock` allows access to the (simulated) clock of the CSA. Finally, `Lance` is used as an interface to the (simulated) network driver and the network module itself.

Note that the `Controller` and `Networks` are global modules, whereas the modules of the CSA are replicated for every used area on every node.

The principles of controlling a module are very simple. Each module has three functions: `SetParameter()` for changing the value of a system parameter stored in the module, `GetParameter()` for reading a certain value, and `ExecuteCommand()` to prompt the module to do something. For example, if the user (through `EvalSys`) wants to read a parameter in module `Clock`, `EvalSys` just has to send the `Supervisor` of the appropriate CSA a message (see the Evaluation Interface Manual for details), and the `Supervisor` calls the `GetParameter()` function of `Clock` and sends back the result to `EvalSys`. In the other direction, for example, if `Clock` wants to report that it has changed the clock value (as requested by the clock synchronization), maybe including the old and new clock value in its report as well, it sends an event to the evaluation using function `NoteEvent()` of the module responsible for it. In the case of a module belonging to a CSA, that would be the `Supervisor` of the CSA, and for the `Network` it is the `Controller`. The event will automatically get timestamped using a global time base that is available to all modules and be sent to `EvalSys` at the next opportunity.

Fig. 5 gives a detailed view of the dependencies between those modules of a CSA that take actively part in the synchronization. In the course of a full message exchange, `ClockSync` hands over a message to the `Lance`, which passes

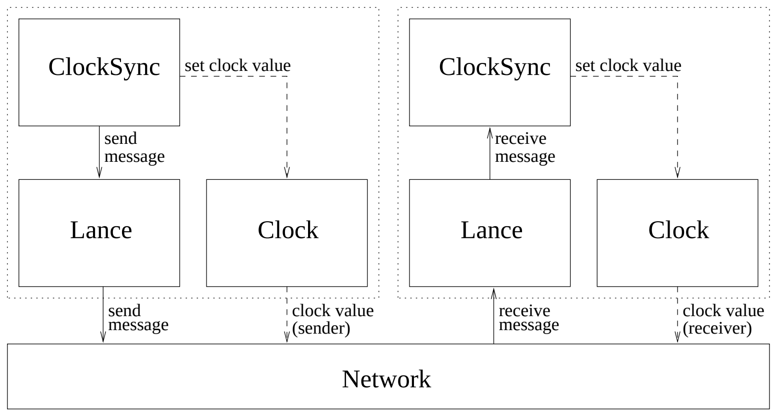

Figure 5: Module Interaction

it on to the `Network`. Here it is delayed to simulate a *broadcast delay* (p.e. caused by medium access conflicts), timestamped on transmission with the local time of the sender node (as is done in hardware by the NTI), and then delayed again for some time to simulate the *transmission delay* before it is timestamped with the clock values of all CSAs of the receiving nodes. The message is then announced to the `Lance` modules of all CSAs on that node, which in turn pass them on to their `ClockSync` modules. At the end of the synchronization round, a `ClockSync` module uses all data collected in that round to compute a new clock value and sets the clock accordingly.

4 Simulation Modules

This section shows how to operate with the modules described in the previous section. For each module, all parameters, commands and events are listed, together with their meaning and effects. Most likely, the evaluation system `EvalSys`[3] will provide a graphical interface to the simulation, so you (the user) will not be troubled with the exact names of the parameters/commands/events listed below. I will also assume that `EvalSys` will shield the user from unnecessary details, so I will not mention which information is needed to set/read a parameter, what you have to specify to execute a command, or list the data that is sent by an event. Neither have I mentioned the default values of the parameters (and up to a certain extent also of commands) that the simulation uses as long as `EvalSys` does not change them. `EvalSys` should make that clear (see [Gri99]). If it does not, or if you desire more knowledge of these elements, see the Evaluation Interface Manual for a complete description of the interface to the evaluation.

[3]References to "`EvalSys`" do *not* mean the dummy evaluation that is part of the `SimUTC` archive, but an independently developed graphical system as we use in our project [Gri99]. However, any interface program will work as long as it conforms to the interface description.

4.1 Controller

This module is the top component of the simulation. It has a connection (socket) to the evaluation and is responsible for the management of applications (CSAs). Furthermore, it handles the network module(s) and is their link to the evaluation, i.e., it sends any events from the `Network` to the evaluation and passes messages from `EvalSys` to the `Network`.

After starting the program, the `Controller` will be the only existing module. You can then create new CSAs or networks by sending commands to the `Controller`. The module periodically checks the socket for new messages from the evaluation. Note that no messages will get lost, since they are buffered until the `Controller` reads them. The simple reason for a certain period is to avoid blocking the CPU.

The `Controller` has only a few *parameters* to control its behaviour:

EventClassReport: controls the report of whole event classes. Every event belongs to a class (see Appendix C for the mapping event $\rightarrow$ class), and this parameter allows the user to turn on/off complete event classes. Please note that this parameter does not influence `EventReport` or vice versa. If you turn off an event class and later on enable it again, all events of that class that you have enabled with `EventReport` will be reported again.

EventReport: controls the report of single events. To avoid flooding the evaluation with events, the user may turn on/off every event separately, thus providing a means to filter out all events that are not of interest.

InterfacePeriod: sets the period of the `Controller`. This determines how often the `Controller` checks the socket to the evaluation (i.e., the reaction time to the evaluation) and the frequency with which event reports from the `Network` are sent to `EvalSys`.

TimerGranularity: is the granularity of the timer that is used for achieving the interface period. The parameter can only be read. When using the pSOS^{+m} timer, the value will be 10^{-2} seconds.

However, the `Controller` does have a number of *commands* to control the execution of the CSAs:

CreateApplication: creates a new CSA, which will be suspended at first, i.e., it does not participate in the synchronization, although it can already accept data from the evaluation.

DeleteApplication: deletes an existing CSA. It is the task of `EvalSys` to notify the other applications that there is now one node less.

SuspendApplication: suspends an existing application. As explained above, the CSA will stop participating in the synchronization and ignore all messages from the other nodes. The currently available synchronization algorithms will also discard any data (even delay- and rate measurements) they have collected (but this can be changed, see Section 5). The CSA will stay in suspension until it is resumed.

ResumeApplication: the pendant to the above command. A suspended application will be resumed. If the application was not suspended, the command has no effect.

SuspendSimulation: like `SuspendApplication`, but it suspends all existing CSAs.

ResumeSimulation: resumes all CSAs. Again, it has no effect on CSAs that were not suspended.

ResetSimulation: resets all CSAs and networks (see the description of command `ResetModule` of the modules concerned).

KillSimulation: deletes all applications, deletes the networks, and exits the program.

CreateNetwork: creates a new network and appoints a unique network identifier to it, which is returned to `EvalSys`.

DeleteNetwork: deletes an existing network. You should take care not to delete networks that are associated to any nodes. Either suspend all CSAs with this associated network, or set their associated network to some other network before you delete it.

ResetModule: restores the `Controller` to its initial state after startup. The command causes the `Controller` to delete all networks and CSAs.

Since the `Controller` does not actively participate in the synchronization, it has no own events.

4.2 Supervisor

The `Supervisor` is like the `Controller` insofar as it maintains its own socket to the evaluation and acts as link to `EvalSys` for all other application specific modules, namely `Clock`, `ClockSync`, and `Lance`. Similar to the `Controller`, it is executed periodically to service the socket and the events from the application modules. In contrast to the `Controller`, it may also initiate system snapshots, i.e., prompt all clocks to read their values simultaneously and send them to the evaluation (through events).

The `Supervisor` has the same *parameters* as the `Controller`:

EventClassReport: controls the report of whole event classes. Has the same behaviour as the corresponding parameter of the `Controller`.

EventReport: controls the report of single events from the CSA modules. Again, it has the same behaviour as the corresponding parameter of the `Controller`.

InterfacePeriod: sets the period of the `Supervisor`. Apart from the response time to the evaluation, this also determines the frequency of system snapshots.

TimerGranularity: is the granularity of the timer that is used for achieving the interface period. The parameter can only be read. When using the pSOS^{+m} timer, the value will be 10^{-2} seconds.

In contrast to the `Controller`, the `Supervisor` has just a few *commands*:

EnableSnapshot: prompts the `Supervisor` to trigger a system snapshot once every period until the feature is disabled again. Usually, one dedicated `Supervisor` takes these snapshots. Suspending the application does not affect this feature.

DisableSnapshot: the `Supervisor` stops making system snapshots. This is the default after creation of the CSA. The command has no effect if the `Supervisor` was not triggering snapshots anyway.

ResetModule: causes the `Supervisor` to reset all modules belonging to its CSA, namely `ClockSync`, `Clock`, `Lance` and itself, so the complete CSA will restore its initial state.

The `Supervisor` does not take part in the synchronization, so it has no own events to report.

4.3 ClockSync

This is the main module of the clock synchronization application, which implements the synchronization algorithm. The *fixed* parts include periodically sending a synchronization message, optionally adding clock rate information to it, receiving the messages from the other nodes, replying to requests for delay measurement, etc. The *exchangeable* parts do all algorithm specific work, i.e., processing the data from the other nodes, doing rate synchronization, computing the transmission delay estimates, and last not least computing the new clock value for adjustment at the end of the resynchronization period. These parts have to be replaced to implement other clock (rate) synchronization algorithms and delay measurement methods (see Section 5).

After startup, the module is suspended, i.e., it does not accept any messages from the `Lance`, nor does it execute its periodic work. Resuming the CSA (through a command to the `Controller`) will resume this module as well.

The synchronization module has plenty of *parameters*:

AssociatedNetwork: sets the associated network of the CSA. Only the synchronization data received from this network is used as input to the convergence function (CV). The network must be connected.

ComputationDelayCompensation: denotes the waiting time from calling the CV until the clock is set to its result. It is a worst case estimate of the execution time of the convergence function. In pure simulation, this execution time is zero, but you can set the parameter to a positive value as desired.

DelayCompensation: sets the waiting time between sending a synchronization message and applying the CV. The time should be large enough to allow all messages from non-faulty nodes to arrive before the CV is executed.

DelayMeasurementCycles: sets the number of rounds from one delay measurement round to the next. In a delay measurement round, the normal clock synchronization message contains a request for delay measurement, and the other nodes will send a reply to that request, containing all data the requesting node needs to compute estimates for transmission delays between its node and the replying nodes.

After creation of the CSA, delay measurement is turned off (i.e., the value is set to zero).

Delta, EpsilonPlus, EpsilonMinus: assign new values to the estimated transmission characteristics. These estimates are the result of delay measurements. The user may read and modify existing entries, but never create new ones[4]. Note that independent access to these parameters is possible (even necessary), although they are grouped together in this description.

Note that the worth of these parameters should not be overestimated. Since `EvalSys` has no direct control on the execution of `SimUTC`, it is virtually impossible to deterministically change the values at a specific round. The

[4]The restriction is needed to avoid problems caused by concurrent modifications of the list that stores these entries. Even though the pure simulation would allow such modifications, portability to the hardware calls for this condition.

greatest use of the parameters lies in providing initial values to the algorithm before the simulation is started (but after the algorithm type is set)[5].

NumberOfNodes: sets the number of nodes (i.e., number of CSAs) in the system. This is an important information for the clock synchronization algorithm.

NumberOfFaultyNodes: sets the maximum number of faulty nodes that are tolerated by the synchronization. Normally, the synchronization algorithm will determine this value[6]. The parameter is important for the synchronization algorithm.

RateSynchronizationCycles: sets the number of rounds from one rate synchronization round to the next. In a rate synchronization round, the normal clock synchronization message carries additional information about the clock rate which enables the receiving nodes to synchronize their rates.

After creation of the CSA, rate synchronization is turned off.

RoundPeriod: controls the period[7] of sending clock synchronization messages to the other nodes. Note that the period is achieved by programming the clock to trigger an interrupt whenever the clock value is a multiple of this parameter. Thus, if all clocks are synchronized and the round periods are the same, the rounds of all CSAs will start at approximately the same time.

SynchronizationAlgorithm: sets the type of the clock synchronization algorithm that is used by the CSA. You may change the algorithm during operation, but be aware that changing the algorithm discards all data that has been collected (measured delays, rates, and collected intervals). In order to allow the algorithm to use valid system parameter values during its initialization, you should set the algorithm type only after you have set all other parameters of `ClockSync` and maybe also `Clock` that the algorithm needs.

See section 5 for details on how to add new algorithm types.

ThetaPlus, ThetaMinus, U: assign new values to the estimated bounds on the local clock drifts and the accumulated state adjustment during a rate synchronization round. The data is needed for rate synchronization, which results in adjusting the rate of a local clock. Again, the parameters are accessed independently.

[5]Of course, this is only true in pure simulation. In hardware-based simulation, you have nearly the whole round period to set the values that are used in this round. In this case, you can also utilize the parameters during operation.

[6]Likely values are $f < n/2$ or even $f < n/3$, with n representing the number of nodes and f the maximum number of faulty nodes.

[7]The period will typically be in the range 10-100 s.

These parameters have the same drawback as the parameters for changing the transmission delay estimations: in pure simulation, it is virtually impossible to set them deterministically during operation.

TimerGranularity: is the granularity of the clock timer that is used for achieving the round period. The parameter can only be read. All period specific values that can be set in module `ClockSync` (e.g. the round period) must be multiples of this granularity. When using the UTCSU, the value will be 2^{-16} seconds.

UserParameter: specifies a parameter that is not processed by this implementation, but passed through to the exchangeable part.

There are only a few *commands*:

ConnectNetwork: adds a network to the list of connected networks. If this is the only connected network, the associated network is automatically set to it.

DisconnectNetwork: removes a network from the list of connected networks. If the network is the associated network, the associated network is set to the next network in the list. If there are no more networks, the CSA now has no associated network.

DontReportTransmissions: tells the module that it should not report measured network transmission delays to `EvalSys`. This is the default.

ReportTransmissions: tells the module that it should report measured network transmission delays to `EvalSys`. Measured transmission delays are only available in hardware-based simulation, and even there only in special test environments, where the nodes are connected to each other in such a way that the transmission of a message on one node can be timestamped on all other nodes. Do not confuse them with the transmission delay estimations that are accessed via `Delta`, `EpsilonPlus`, `EpsilonMinus`.

In pure simulation, the command has no effect.

ResetModule: restores the initial state of the module.

UserCommand: is passed on to the exchangeable part of the synchronization module.

At last we have a module with *events*:

ChangeAssociation: is sent if the associated network is changed by the synchronization algorithm (p.e., when the first network is connected).

ConvergenceFailed: is sent if the clock synchronization algorithm (convergence function) was invoked, but for some reason could not compute a new clock value.

DelayEstimation: is sent whenever `ClockSync` receives a reply to a delay measurement message and estimates the transmission delay from the sender node to its own node. The estimation is typically based on the delay of the message itself and on previously collected transmission delay data. Its implementation belongs to the exchangeable part of the module and may depend on the synchronization algorithm that is installed. The values sent in this event can also be obtained through the `Delta`, `EpsilonPlus` and `EpsilonMinus` parameters.

IntervalEvent: contains either an input interval to the CV, or the result of the CV. The input intervals are sent when they are received by the CSA, the resulting interval is sent after the convergence function has been executed.

RateEvent: contains either input data to the rate synchronization algorithm, or its result. Most of the currently available algorithm types do not send this event.

ReceivedMessage: reports that a message was received.

SendMessage: reports that a message was sent (or at least passed to the network driver).

TimerEvent: is sent whenever the clock timer that is used for achieving the round period generates an interrupt.

TransmissionDelay: is sent whenever `ClockSync` receives a message and if it should report measured network transmission delays. In this case, the delay is reported in this event. The delay is the time from sending the message until the message is timestamped at the receiver, measured with the clock of the receiver. It is only available in special hardware environments, never in pure simulation.

UserEvent: some event from the exchangeable part of the synchronization.

4.4 Clock

This module simulates an adder-based clock and forms its interface. It allows atomic read/write access to the clock value and implements the continuous amortization technique, which is the way how the clock attains a new value set force by the clock synchronization algorithm. More precisely, the current clock rate is modified appropriately such that the new clock value is reached within a certain amount of time.

There is nearly no difference between a `Clock` module with or without connected GPS receivers, making the distinction between P- and S-Nodes, see Section 2. Internally, a module with GPS refuses to take over a new clock value that is computed by `ClockSync` in the course of the clock synchronization, but approximately keeps track to the GPS time instead. To the user, it behaves like the `Clock` module of an S-Node with an additional event.

In the course of fault injection, the `Clock` may crash (that is, the clock stops counting), it may return arbitrary values for its time registers or set them to arbitrary values upon a write access, and it may change its drift. Faults can be injected whenever the clock time is read or set by the CSA.

There are several *parameters* that may be accessed:

Accuracy: allows atomic access to the accuracy value of the clock. When reading it, the four byte parameter is split into two halfs containing $\alpha^{\pm}$. For setting the accuracy, you can specify a signed four byte value, and both accuracies are set to that value[8].

This parameter directly accesses the clock registers, so it is not affected by fault injection.

AccuracyTransmissionLoss: specifies the granularity of the accuracy values transmitted over the network. Most likely, it will be 2^{-23} seconds and you will not need to bother with it.

AmortizationDuration: The duration of the amortization phase. The new clock value will be attained in this time.

Here, we have a slight difference between P- and S-Nodes: since in the worst case a P-Node tries to synchronize to GPS every second, the duration of the amortization phase should be less than one second. Thus, P-Nodes refuse to accept values that are greater or equal to one second, whereas S-Nodes accept all non-negative values.

ClockDrift: sets the rate drift of the clock, i.e., the deviation from the ideal clock rate. This parameter is used to control the drift of the clock which is due to an imperfect oscillator in hardware. Note that the synchronization algorithm has no access to this information, so there is barely any difference between an imperfect hardware clock and a software clock with a non-zero drift.[9]

The parameter is only available in pure simulation.

[8]Remember that in case of the UTCSU, the value corresponds to bits S:[-8,-38], whereas the accuracies that are read correspond to bits [-8,-23]. So, as you can see here, the internal resolution of the clock is much better than its external resolution.

[9]By the way: normal clock oscillators have a drift around 10^{-5}, good oscillators get that down to 10^{-7}. Atomic clocks can go as low as 10^{-11}.

ClockGranularity: is the granularity of the clock time value that is transmitted over the network. Normally, it will be 2^{-24} seconds and you will not need to change it.

ClockValue: gives atomic access to the macrostamp and timestamp value of the clock. If you set the parameter, the clock is directly set to this value; no amortization is used.

The parameter directly accesses the clock registers, so it is not affected by fault injection.

CrashStatus: this parameter is read-only and returns the current status of the module.

DistributionFast: denotes the distribution data of faulty clock drifts within the interval (ρ^+, ρ^+_{wc}), where ρ^+ is the upper bound on the non-faulty drift (see parameter `RhoPlus`), and ρ^+_{wc} is the upper bound on the faulty drift.

DistributionRegular: denotes the distribution data of faulty clock drifts within the interval $(-\rho^-, \rho^+)$, where $\rho^\pm$ are the bounds on the non-faulty drift. The bounds are set with parameters `DistributionFast` and `DistributionSlow`. `DistributionRegular` only sets the rest of the distribution data.

DistributionSlow: denotes the distribution data of faulty clock drifts within the interval $(-\rho^-_{wc}, -\rho^-)$, where ρ^- is the lower bound on the non-faulty drift (see parameter `RhoMinus`), and ρ^-_{wc} is the lower bound on the faulty drift.

PCrash, PRecover: sets the probability that the module crashes or recovers. A crashed `Clock` will stand still and ignore all write accesses to its clock registers. Reading the clock returns the frozen clock value.

PDriftChange: sets the probability that the drift of the clock changes during fault injection.

PFast, PSlow: sets the probability that a faulty drift is faster or slower than the expected drift bounds allow. The parameters are independent.

PReadFault, PWriteFault: sets the probability that reading the clock time returns arbitrary values, or that setting the clock time sets arbitrary values. The parameters are independent.

RhoPlus, RhoMinus: these read-only parameters return the worst case (upper or lower) bound on the non-faulty clock drift. The bounds are set with parameters `DistributionFast` and `DistributionSlow`. Although they are listed together, the parameters are independent.

WobbleDrift: sets all data necessary to incrementally change the clock drift over a given range, which must be within the bounds for the non-faulty clock drift. The idea is to let the clock drift increase until it reaches a given upper bound and then let it decrease until it reaches the lower bound, where it starts to increase again. The drift periodically changes by a value that is obtained from the distribution data you specify along with the parameter. Each period, a new incremental value is obtained and added to the clock drift. To use this feature, you have to specify a positive period. Setting the period to zero stops drift wobbling. If you want to set the drift back to zero before stopping, you should leave the period on a positive value and set the wobble range bounds to zero instead.

Note that if you change the bounds of the non-faulty drift after you have set the wobbling range, then the range will be adapted to conform to the new bounds.

Several *commands* are possible:

ConnectGps: notifies the `Clock` that there is a GPS receiver connected to it. The command prepares the module to handle the time information data from the GPS receiver. If the command is executed for the first time, the node type will automatically change from S-Node to P-Node.

DisconnectGps: Disconnects a formerly connected GPS receiver, i.e., its time information data will not be used. If the last receiver of this CSA is disconnected, the node type will automatically change from P-Node to S-Node.

CrashModule: Crash the `Clock`.

RecoverModule: Recover a crashed `Clock`.

ResetModule: restores the initial state of the module.

And `Clock` may send these *events*:

ClockPreload: is sent if the clock is preloaded to a new value (with amortization kicking in at some future time).

ClockReadFault: the CSA has tried to read the clock time, but the clock has returned arbitrary values instead of the correct time.

ClockUpdate: is sent if the clock is set to a new value (amortization starts at once).

ClockWriteFault: the CSA has tried to set the clock time, but the clock has set arbitrary values instead of the requested time.

CrashStatusChange: the crash status of the `Clock` has changed.

GpsEvent: is only used by P-Nodes. After the GPS time was read, it is compared to the clock value. If there is a significant difference, the clock value is set to the GPS time, and this new value will be reported with this event.

IgnoreSetting: the CSA has tried to access some clock register, but the clock was crashed and has ignored this access.

Snapshot: sends the clock value sampled during a system snapshot.

TimerEvent: is sent whenever the clock timer that is used for achieving the period for drift wobbling generates an interrupt.

4.5 Lance

This module is a simple interface to the network driver. It does not do much except passing messages to the network calling the simulated CI (Clock Interface) functions of the network driver (see [Ri97] for more information about the driver) and delivering received messages to the `ClockSync` module. It may scramble, duplicate, delay and/or discard messages at will.

After startup, the module is suspended and does not accept any messages from the `Network`. This state is left with the first message that is sent from `ClockSync`.

The module has a few *parameters*:

CrashStatus: this parameter is read-only and returns the current status of the module.

DistributionRecv: sets the distribution data for choosing the delay of incoming messages.

DistributionSend: sets the distribution data for choosing the delay of outgoing messages.

PCrash, PRecover: sets the probability that the module crashes or recovers. A crashed `Lance` will discard all messages that are currently being delayed and will discard all messages that are sent or received.

PDuplicateRecv: sets the probability that an outgoing message is duplicated. This means that it is sent twice.

PDuplicateSend: sets the probability that an incoming message is duplicated.

PScrambleRecv: sets the probability that an outgoing message is scrambled. Scrambling a message means that all its data bytes are XORed to a random value and the result is used as the new message.

PScrambleSend: sets the probability that an incoming message is scrambled.

Some *commands* are available:

CrashModule: crashes the `Lance`.

RecoverModule: recovers a crashed `Lance`.

ResetModule: restores the initial state of the module (suspended).

And it may send these *events*:

CrashStatusChange: the crash status of the `Lance` has changed.

DiscardSendMessage: a message was discarded by the sender `Lance`.

DiscardRecvMessage: a message was discarded by the receiver `Lance`.

DuplicateSendMessage: a message was duplicated by the sender `Lance`.

DuplicateRecvMessage: a message was duplicated by the receiver `Lance`.

ScrambleSendMessage: a message was scrambled by the sender `Lance`.

ScrambleRecvMessage: a message was scrambled by the receiver `Lance`.

TimerEvent: is sent whenever the clock timer that is used for delaying messages generates an interrupt.

4.6 Network

The `Network` simulates the transmission of messages over a network. It accepts messages from the `Lance`s, randomly delays them according to the broadcast latency of the network, and then transmits the messages to all receiver nodes, i.e., (randomly) delays each dedicated message according to the transmission characteristics stored for the pair $\langle senderNode,\ receiverNode\rangle$.

There may be more than one network at the same time, and the types of the networks (broadcast, point-to-point, etc.) may vary. Right now, the simulation encompasses just a broadcast network. See Appendix 6 for instructions on what to do if another type of network is needed.

The network allows access to several *parameters*:

PCrash, PRecover: sets the probability that the module crashes or recovers. A crashed network will discard all messages that are currently being transmitted and refuse to transmit new messages.

CrashStatus: this parameter is read-only and returns the current status of the module.

PEarly, PLate: sets the probability that messages transmitted between a pair of nodes arrive earlier or later than the transmission characteristics would allow.

DistributionTimely: sets the distribution data for choosing a transmission delay for a message that is transmitted between a pair of nodes and where the transmission time conforms to the transmission characteristics. The specified delay interval corresponds to $(\delta - \epsilon^{-}, \delta + \epsilon^{+})$.

Note that the network will only transmit messages between an ordered pair $\langle senderNode,\ receiverNode\rangle$ if both nodes are connected and if transmission delay data is stored for the pair. The only exception is if the sender and receiver are the same: in this case, if the node is connected and no data is stored, then the message will be transmitted with a constant delay.

Since the data of this parameter influences the bounds for early and late messages, you should always set the initial delay data in the order `DistributionLate` before `DistributionTimely` before `DistributionEarly`.

DistributionEarly: sets the distribution data for choosing a transmission delay for a message that arrives sooner than the transmission characteristics would allow. The upper bound on the delay is $\delta - \epsilon^{-}$, which is derived from the data specified by `DistributionTimely` and overrides the bound set by this parameter.

Note that during the initial configuration you should always set `DistributionTimely` before setting `DistributionEarly`.

DistributionLate: sets the distribution data for choosing a transmission delay for a message that arrives later than specified through the transmission characteristics. The lower bound on the delay is $\delta + \epsilon^{+}$, which is derived from the data specified by `DistributionTimely` and overrides the bound set by this parameter. During the initial configuration, set this parameter before setting `DistributionTimely`.

DistributionLambda: sets the distribution data that is used for choosing the value of the broadcast latency λ for a specific message. The broadcast latency is the time from the send request of the node to the point where the message is actually put on the network.

DistributionOmega: sets the distribution data for choosing the broadcast operation delay ω of a message. The broadcast operation delay is important for point-to-point networks. It specifies the time from the transmission of a broad- or multicast message to the first node in the set of receivers until the transmission of that message to the last node in the set.

The user can request the execution of a few *commands*:

ConnectNode: connect a new node to the network (that is, enable the node to send messages to all other nodes connected to that network, and to receive messages from them). If the node was already connected, the command has no effect.

Note that the description is only valid for the implemented "broadcast" network. If other networks are implemented, the semantic may vary.

DisconnectNode: remove the node from the list of connected networks. If the node tries to transmit a message to the network, this message will be discarded. Furthermore, no other node will be able to send a message to an unconnected node. If the node was not connected, the command has no effect.

Again, this behaviour may change for other networks.

CrashModule: crash the `Network`.

RecoverModule: recover a crashed `Network`.

ResetModule: restore the initial state of the network. The command discards all transmission data and disconnects all connected nodes.

The network sends some *events*:

CrashStatusChange: the crash status of the `Network` has changed.

SendMessage: a message was transmitted (after the initial broadcast delay). The message will arrive in time.

EarlyMessage: a message was transmitted. The message will arrive early.

LateMessage: a message was transmitted. The message will arrive late.

DeliverMessage: a message was delivered.

MessageLost: a message was discarded.

5 Adding New Algorithm Types

A CSA contains three user definable parts:

- delay measurement
- rate synchronization
- clock synchronization algorithm

All three have to be implemented in a class that should be derived from the abstract class `ACClockSyncAlgorithm` in file `algbase.hpp`.

SimUTC already includes some algorithms. You may choose one of the following algorithm types:

- FreeRun: there is no synchronization at all, the clocks are allowed to drift.
- FaultTolerantAverage: is described below.
- FaultTolerantIntersection: is described below.
- Fetzer: like `FaultTolerantAverage`, but it handles data from P-nodes differently (Fetzer algorithm).
- Marzullo: is described below.
- Midpoint: like `FaultTolerantAverage`, but it uses the midpoint of the intervals instead of the reference point.
- OP: is described below.
- Rate: is described below.

FaultTolerantAverage

- The clock synchronization algorithm utilizes the well-known *Fault Tolerant Average* (FTA) as convergence function. As input, it uses only the data from the associated network. If a new timestamped message is received, the algorithm computes a clock time interval and a single timestamp in it (the reference point) using the formula given in the description of the generic algorithm in [SS97]. This interval and reference point is stored until the convergence function is called. The CV will go through the list of intervals and use the reference points as input[10]. These values are sorted, the first and last f are discarded, and the remaining ones are averaged to obtain the new clock time value. For the sake of simplicity, the accuracies are constantly zero. After the computation, the list of intervals is emptied for the next round. The clock value is stored until the clock is resynchronized.

[10] Here we have the only difference to algorithm type `Midpoint`, which uses the midpoints of of the intervals, not the reference points.

- There is no rate algorithm implemented, so all clocks are assumed to have worst case drift bounds.

- The delay measurement strategy is very simple: assuming that there was some transmission delay estimate $(\delta_{k-1}, \epsilon^{\pm}_{k-1})$ stored and that $[d^-_k, d^+_k]$ denotes the interval which contains half of the round trip delay measured in period n, then the new estimate is obtained through $\delta_k = mean(\delta_{k-1}, mean(d^-_k, d^+_k))$ and $\epsilon^{\pm}_k = max(\epsilon^{\pm}_{k-1}, |d^{\pm}_k - \delta_{k-1}|)$.

You will find these implementations in class `CFta` (file `algfta.hpp`). It can serve as an example on how to create your own class. The delay measurement strategy is implemented in the base class, `ACClockSyncAlgorithm`.

FaultTolerantIntersection

- The clock synchronization algorithm utilizes the *Fault-Tolerant Intersection* (FTI) as convergence function (yet unpublished). As input, it uses only the data from the associated network. If a new timestamped message is received, the algorithm computes a clock time interval and a single timestamp in it (the reference point) using the formula given in the description of the generic algorithm in [SS97]. This interval and reference point is stored until the convergence function is called. The CV will go through the list of intervals and computes the compatible accuracy intervals as an array of type `SAsymInterval`. The FTI function is applied on the accuracy intervals, where the reference point is set to the midpoint, leading to the new accuracy interval. After the computation, the list of intervals is emptied for the next round. The clock value and interval of accuracies are stored until the clock is resynchronized.

- There is no rate algorithm implemented, so all clocks are assumed to have worst case drift bounds.

- In order to achieve a good setting of the transmission delay parameters, we make a few (given by `StartSynchronization`) transmission delay measurements before the algorithm starts using them. The strategy itself is the same as that used for `FaultTolerantAverage`.

Marzullo

- The clock synchronization algorithm utilizes the *Marzullo's function* as convergence function, see [Mar84]. As input, it uses only the data from the associated network. If a new timestamped message is received, the algorithm computes a clock time interval and a single timestamp in it (the reference point) using the formula given in the description of the generic algorithm in [SS97]. This interval and reference point is stored until the convergence function is called. The CV will go through the list of intervals and computes the compatible accuracy intervals as an array of type `SAsymInterval`. The Marzullo function is applied on the accuracy intervals, where the reference point is set to the midpoint, leading to the new accuracy interval. After the computation, the list of intervals is emptied for the next round. The clock value and interval of accuracies are stored until the clock is resynchronized.

- There is no rate algorithm implemented, so all clocks are assumed to have worst case drift bounds.

- The algorithm uses the same delay measurement strategy as `FaultTolerantIntersection`.

OP

- The clock synchronization algorithm utilizes the *Optimal Precision* (OP) as convergence function, see [Sch97]. As input, it uses only the data from the associated network as well as precomputed parameters. If a new timestamped message is received, the algorithm computes a clock time interval and a single timestamp in it (the reference point) using the formula given in the description of the generic algorithm in [SS97]. This interval and reference point is stored until the convergence function is called. The CV will go through the list of intervals and computes the associated precision intervals and the compatible accuracy intervals, both as an array of type `SAsymInterval`, where at index 0 the own ones are stored. The Marzullo function is applied on the precision intervals, the resulting interval is intersected with the own one, and the reference point is fixed, leading to the new precision interval (hence reference point). Similary, the Marzullo function is applied on the accuracy intervals, the resulting interval is intersected with the own one, and extended to contain the new reference point, leading to the new accuracy interval. After the computation, the list of intervals is emptied for the next round. The clock value and interval of accuracies are stored until the clock is resynchronized.

- There is no rate algorithm implemented, so all clocks are assumed to have worst case drift bounds.

- The algorithm uses the same delay measurement strategy as `FaultTolerantIntersection`.

Rate

- The clock state synchronization algorithm utilizes the *Optimal Precision* (OP) as convergence function, see [Sch97]. As input, it uses only the data from the associated network as well as precomputed parameters. If a new timestamped message is received, the algorithm computes a clock time interval and a single timestamp in it (the reference point) using the formula given in the description of the generic algorithm in [SS97]. This interval and reference point is stored until the convergence function is called. The CV will go through the list of intervals and computes the associated precision intervals and the compatible accuracy intervals, both as an array of type `SAsymInterval`, where at index 0 the own ones are stored. The Marzullo function is applied on the precision intervals, the resulting interval is intersected with the own one, and the reference point is fixed, leading to the new precision interval (hence reference point). Similary, the Marzullo function is applied on the accuracy intervals, the resulting interval is intersected with the own one, and extended to contain the new reference point, leading to the new accuracy interval. After the computation, the list of intervals is emptied for the next round. The clock value and interval of accuracies are stored until the clock is resynchronized.

- The clock rate synchronization algorithm utilizes the *Optimal Precision* (OP) as convergence function, see [Sch97]. As input, it uses only the data from the associated network as well as precomputed parameters. If a new timestamped message is received, the algorithm computes a quotient rate interval (relative rate measurement), followed by the remote rate interval using the formula given in the description of the generic algorithm in [Scho97]. This interval and reference point is stored until the convergence function is called. The CV will go through the list of intervals and computes the associated consonance intervals and the rate intervals, both as an array of type `SAsymInterval`, where at index 0 the own ones are stored. The Marzullo function is applied on the consonance intervals, the resulting interval is intersected with the own one, and the reference point is fixed, leading to the new consonance interval (hence reference point = multiplication factor for register STEP). Similary, the Marzullo function is applied on the rate intervals, the resulting interval is intersected with the own one, extended to contain the new reference point, and normalized to account for the rate correction, leading to the new rate interval. The latter is used instead of the worst case drift bounds. After the computation, the list of intervals is kept up to enable the next relative rate measurement. The clock value and interval of accuracies are stored until the clock is resynchronized.

- The algorithm uses the same delay measurement strategy as `FaultTolerantIntersection`.

For implementing your own algorithms, you should first have a look at your base class in file `algbase.hpp` (directory `SimUTC/Sources`) and at the corresponding documentation file `algbase.html` (directory `SimUTC/Doc/WWW`). There, you will find the function declarations of the class and their descriptions.

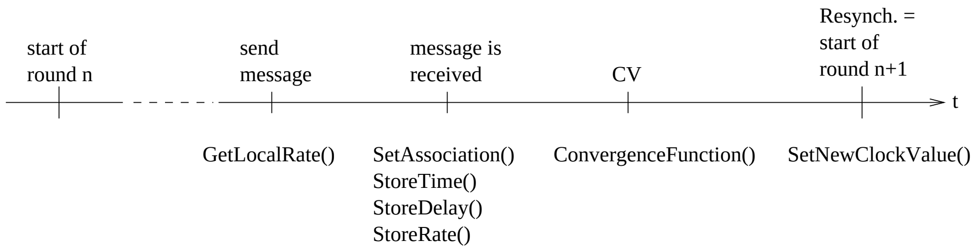

Figure 6: Function Calls to `ACClockSyncAlgorithm`

Fig. 6 shows which functions are called at which stages of the synchronization round. Please read the documentation `algbase.html` for further information about the functions.

Of course, you must implement the pure virtual functions to obtain a working class. You may overload any other virtual function if you need different behaviour. You should not overload non-virtual functions, since the fixed part of `ClockSync` uses your class through a pointer of type `ACClockSyncAlgorithm`.

In any case, you must provide a static member function with the prototype `ACClockSyncAlgorithm *YourFunction (CClockSync *)`, which should simply create and return a new object of your class type. In the program files, this function is called the "create function". The simulation uses it to obtain a new object of your class. You may choose an arbitrary name for this function.

If you wish to do initial computations in the constructor of your class and if you need data from other modules like `ClockSync`, you must ensure that `EvalSys` sets your algorithm only after it has configured all other parameters of `ClockSync` (and possibly of `Clock` if you need the clock as well). The documentation to the evaluation interface [Gri99] advises the programmer of `EvalSys` to call parameter `SynchronizationAlgorithm` last anyway, but if there are any problems, you should check your particular implementation of `EvalSys`. The dummy evaluation that is distributed along with `SimUTC` sets the algorithm type last.

Another problem that may occur if you need any modules in the constructor of your algorithm is related to your choice of the default algorithm. The default algorithm is set to FreeRun in `ClockSync`, but if you change this, then you have to keep in mind that the default algorithm is already created during the creation of the CSA. But at this time, the `Clock` module is still not connected to `ClockSync`. Therefore, if you need clock data for your computations in the constructor, you should always check if the clock pointer you obtain from inherited member function `GetClock()` is not NULL. Only assert that the pointer is not

NULL if you are sure that your algorithm is not the default algorithm, otherwise just do not execute your computations. They will be executed when `EvalSys` sets the algorithm, because at this point the clock is already available.

Your interface to the evaluation program consists of just four functions: `SetParameter()` and `GetParameter()` are called whenever the evaluation needs access to a parameter. `ExecuteCommand()` is called if the evaluation wants to execute a command, and you can call `ReportEvent()` if you wish to report an event.

If you need additional parameters, commands or events, you can overload any of the first three functions (but not the last one, see below) and use your own structures to pass user-defined data between your class and the evaluation. Make sure that you derive your own structures from the base structures named below, and that all member variables of your own structures are public.

- For adding a new *parameter*, you should use a structure derived from `SParameter` (`structs.hpp`) and assume that the parameter type member of this base structure is set to `UserParameter` (`params.hpp`). Add your own (sub-)parameter type to the derived structure to identify the type of your structure, and add the data members that you need. Overload functions `SetParameter()` and `GetParameter()`, and check if the parameter type of the base structure is `UserParameter`. If it is, you can cast the function argument (of type `SParameter`) to your own structure and process it further. Otherwise, you should call `ACClockSyncAlgorithm::SetParameter()` or `ACClockSyncAlgorithm::GetParameter()` to pass the parameter to your base class. In that case, you should return the value of this function.

- Likewise, to add own *commands*, derive a structure from `SCommand` (`structs.hpp`) and assume that the type member of the base class is set to `UserCommand` (`params.hpp`). Overload function `ExecuteCommand()` and check the type of the `command` argument. If it is `UserCommand`, you can cast the argument to your own type and process it. If it is some other type, pass the command to the base class calling `ACClockSyncAlgorithm::ExecuteCommand()` and return the value of this function.

- If you wish to add new *events*, you should derive a structure from `SApplicationEvent` (`structs.hpp`). Just create a new event of that type, fill in your own data, and use function `ACClockSyncAlgorithm::ReportEvent()` to send the event to `EvalSys`. The parameters of the base structure will be filled in automatically. Note that there is no need to overload any functions for adding new events. Instead, you have to create an object of your event structure and pass it to an existing function. Do not delete the event object, since it is stored until it can be passed on to `EvalSys`. The

Supervisor of your CSA will take care of deleting the event, so you should not use the pointer to the event after you have reported it.

Note that you will have to tell EvalSys how it should actually use new parameters, commands and events. The evaluation system that is currently under development will provide the user with the means to add new parameters, commands and events during operation. See the documentation of EvalSys [Gri99] for further information.

If you wish to add debug messages to your code, you should use one of the macros `Report()`, `Check()` or `Assert()` that are defined in file `error.hpp`. The macros use one of the global error report functions `ReportError()` to print error messages, and these use an (exchangeable!) error handler to direct the output either to the standard output or to some other device. Thus, just by using another error handler, the output can be redirected without any changes within the application. Moreover, most of your code will be executed in the context of an ISR, so you cannot use `printf` or the C++ output streams anyway. The error handler, on the other hand, is executed in another context and does not have this restriction. Therefore, you should never use any other means for your debug output.

Assuming that you have derived and implemented your own class, you will have to add your class to the algorithms available, which is done in several steps:

1. Add a new item for your algorithm type to the enumeration `EAlgorithmType` in file `general.hpp`.

2. Add your create function to the list of installed algorithm create functions. You need to edit the global function `InstallAlgorithms()` in file `globals.cpp`. Follow the instructions given in the comments to this function. Do not forget to `#include` your header file.

3. Add your object file to the list of project files in the `Imakefile` found in directory `SimUTC/Sources` and call `makeit` in directory `SimUTC` to rebuild the project makefile.

4. Change into the source code directory and call `make` to recompile and relink the project. The new executable is already capable of using your algorithm class.

5. Configure EvalSys to accept the new enumeration value that you have added for your algorithm type (see the documentation on EvalSys [Gri99]).

Since SimUTC is a complex tool, implementation errors of the algorithm may result in unexpected behaviour and be quite hard to trace. So I recommend testing your class before you execute the whole simulation. I have used separate test programs (see directory `SimUTC/Sources/Tests`, file `algtest.cpp`) for verifying the correctness of all member functions, and I strongly advise you to do likewise.

6 Adding New Network Types

Adding new network types is more or less like adding new algorithms. You have to derive from a given base class and have to modify the simulation code to install your network.

The base class for all networks is the abstract class `ACNetwork`[11]. You might also take a look at the implemented broadcast network, whose declaration and documentation can be found in `network.hpp` (source code directory) and `network.html` (file description directory).

Class `ACNetwork` provides the basic features all networks are supposed to have, like delivering messages to a specific node after a certain delay, or delaying a message before it is even processed by the network. It also stores the two parameters which describe the behaviour of the network, namely λ (the broadcast latency) and ω (the broadcast operation delay).

You must implement the pure virtual function `SendMsg()` to obtain a working class. You may overload any other virtual function if you need different behaviour. Do not overload non-virtual functions, since all objects using the network use your class through a pointer of type `ACNetwork` and will always use the function of the base class.

In any case, you must provide a static member function with the prototype `ACNetwork *YourFunction (CControllingObject *, SINT32)`, which should simply create and return a new object of your class type. In the program files, this function is called the "create function". The simulation uses it to obtain a new object of your class. You may choose an arbitrary name for this function.

If you need some additional parameter, command, and event structures, you should derive them from `SNetworkParameter`, `SNetworkCommand` and `SNetworkEvent`, respectively. Adding your own structures is much like adding user-defined structures, so see Section 5 for the necessary steps. You can find all structures in file `structs.hpp`.

Let us assume that you have derived and implemented a new network. Then, you need to follow these steps:

1. Add a new item for your network type to the enumeration `ENetworkType` in file `general.hpp`.

[11] See file `SimUTC/Sources/netbase.hpp` for the declaration, and file `SimUTC/Doc/WWW/-netbase.html` for futher explanations of the class.

2. Add your create function to the list of installed network create functions. You need to edit the global function `InstallNetworks()` in file `sim_glob.cpp`. Follow the instructions given in the comments to this function. Do not forget to `#include` your header file.

3. Add your object file to the list of project files in the `Imakefile` found in directory `SimUTC/Sources` and call `makeit` in directory `SimUTC` to rebuild the project makefile.

4. Change into the source code directory and call `make` to recompile and relink the project. The new executable is already capable of using your network class.

5. Configure `EvalSys` to accept the new enumeration value that you have added for your network type (see the documentation on `EvalSys` [Gri99]).

`SimUTC` already offers one network type. This network is more or less a broadcast network in that it can send messages either to all connected nodes or to one dedicated recipient. However, to make the network behaviour more interesting, each message is transmitted independently from all others. So if you send a broadcast, the message will not only arrive at the receivers at different times, but it can even happen that the same or another node sends another message during this transmission and that this second messages arrives sooner than the first. What is more, since all messages are independent and broadcast messages are split into single messages to dedicated receivers, fault injection can cause broadcast messages to be only delivered to a subset of all connected nodes.

Of course, a real broadcast network would only allow to transmit one message at any time (thus delaying the second message until the first is delivered), and it would deliver the message to all receiver nodes at the same time. During fault injection, the message would either be delivered to all nodes or to none.

Implementing a correct broadcast network does not take much effort if you use the currently available network as a reference, so it could be a good exercise before you implement a more complicated network.

7 Future Work

This text describes version 2.3.1.0 of the project SimUTC, which enables the user to execute clock synchronization algorithms with integrated fault-injection, but does not support repeated simulation runs.

This version of the program is already sufficient to subject different clock synchronization algorithms to certain environmental conditions and to monitor their behaviour.

Still unsolved is the problem to provide a *replay* mode. In that version, the user will be able to repeat a simulation run with slight modifications (e.g., exchanging the order of the arrival of two messages on a node) to trace certain actions.

A List of All Commands

The following commands can be sent to all modules in parentheses.

ConnectGps (`Clock`): connect a GPS receiver to the clock

ConnectNetwork (`ClockSync`): connect a network to the CSA

ConnectNode (`Network`): connect a node to the network

CrashModule (`Clock`, `Lance`, `Network`): crash the module

CreateApplication (`Controller`): start a new synchronization application

CreateNetwork (`Controller`): create a new network (pure simulation only!)

DeleteApplication (`Controller`): kill an existing CSA

DeleteNetwork (`Controller`): delete a network (pure simulation only!)

DisableSnapshot (`Supervisor`): disable the triggering of periodic snapshots

DisconnectGps (`Clock`): disconnect a GPS receiver from the clock

DisconnectNetwork (`ClockSync`): disconnect a network from the CSA

DisconnectNode (`Network`): disconnect a node from the network

DontReportTransmissions (`ClockSync`): do not send transmission delay events

EnableSnapshot (`Supervisor`): trigger a snapshot once every period

KillSimulation (`Controller`): delete everything, stop program execution

RecoverModule (`Clock`, `Lance`, `Network`): recover the module

ReportTransmissions (`ClockSync`): send transmission delay events

ResetModule (all modules): restore the initial state

ResetSimulation (`Controller`): reset all application modules (in hardware-based simulation: all CSAs of the node) and all networks (only pure simulation)

ResumeApplication (`Controller`): resume operation of the CSA

ResumeSimulation (`Controller`): resume all applications (in hardware-based simulation: all CSAs of the node)

SuspendApplication (`Controller`): suspend an application (stop operation)

SuspendSimulation (`Controller`): suspend all applications (in hardware-based simulation: all CSAs of the node)

UserCommand (`ClockSync`): user defined command for the synchronization algorithm

B List of All Parameters

This is a list of all system parameters the user can modify to influence the behaviour of the system. The module in parentheses hosts the value.

Accuracy $\alpha^{\pm}$ (`Clock`)
AccuracyTransmissionLoss G_A (`Clock`)
AmortizationDuration t_{amort} (`Clock`)
AssociatedNetwork (`ClockSync`)
ClockDrift ρ (`Clock`)
ClockGranularity G (`Clock`)
ClockValue (`Clock`): clock time value (ms, ts)
ComputationDelayCompensation Γ (`ClockSync`)
DelayCompensation $\Delta + \Lambda + \Omega$ (`ClockSync`)
DelayMeasurementCycles (`ClockSync`)
Delta, EpsilonPlus, EpsilonMinus (`ClockSync`): estimated transmission delay characteristics (δ, $\epsilon^{\pm}$)
DistributionEarly (`Network`): distribution data for choosing the transmission delay in $(D_{min}, \delta - \epsilon^{-})$
DistributionFast (`Clock`): distribution data for choosing a clock drift in $(\rho^{+}, \rho^{+}_{wc})$
DistributionLambda (`Network`): distribution data for choosing the broadcast latency λ
DistributionLate (`Network`): distribution data for choosing the transmission delay in $(\delta + \epsilon^{+}, D_{max})$
DistributionOmega (`Network`): distribution data for choosing the broadcast operation delay ω
DistributionRecv (`Lance`): distribution data for choosing the additional delay of incoming messages
DistributionRegular (`Clock`): distribution data for choosing a clock drift in $(-\rho^{-}, \rho^{+})$
DistributionSend (`Lance`): distribution data for choosing the additional delay of outgoing messages
DistributionSlow (`Clock`): distribution data for choosing a clock drift in $-(\rho^{-}, \rho^{-}_{wc})$
DistributionTimely (`Network`): distribution data for choosing the transmission delay in $(\delta - \epsilon^{-}, \delta + \epsilon^{+})$
EpsilonMinus (`ClockSync`): see `Delta`
EpsilonPlus (`ClockSync`): see `Delta`
EventClassReport (`Controller`, `Supervisor`): events from these classes are reported if they are set in `EventReport`
EventReport (`Controller`, `Supervisor`): events that should be reported
InterfacePeriod (`Controller`, `Supervisor`): socket polling period

NumberOfFaultyNodes f (`ClockSync`)
NumberOfNodes n (`ClockSync`)
PCrash (`Clock`, `Lance`, `Network`): probability of a module crash
PDriftChange (`Clock`): probability of a clock drift change
PDuplicateRecv (`Lance`): probability of duplicating an incoming message
PDuplicateSend (`Lance`): probability of duplicating an outgoing message
PEarly (`Network`): probability of a message arriving too early
PFast (`Clock`): probability of the clock drift changing to a fast drift
PLate (`Network`): probability of a message arriving too late
PReadFault (`Clock`): probability of reading an arbitrary time from the clock
PRecover (`Clock`, `Lance`, `Network`): probability of recovery of a crashed module
PScrambleRecv (`Lance`): probability of scrambling an incoming message
PScrambleSend (`Lance`): probability of scrambling an outgoing message
PSlow (`Clock`): probability of the clock drift changing to a slow drift
PWriteFault (`Clock`): probability of writing an arbitrary time to the clock
RateSynchronizationCycles (`ClockSync`)
RoundPeriod P (`ClockSync`)
SynchronizationAlgorithm (`ClockSync`)
ThetaMinus, ThetaPlus (`ClockSync`): estimated bounds on the clock rate drifts $\theta^{\pm}$
U (`ClockSync`): applied clock state corrections U during a rate round
WobbleDrift (`Clock`): drift wobbling data

There are some parameters that can only be read:

CrashStatus (`Clock`, `Lance`, `Network`): crash status of the module
RhoMinus, RhoPlus (`Clock`): rate drift bounds $\rho^{\pm}$
TimerGranularity (`Controller`, `Supervisor`, `ClockSync`): clock timer granularity G_{CT}

C List of All Events

A list of all events the simulation may send. The module in parentheses is the originator of the event.

ChangeAssociation (`ClockSync`): the associated network was changed
ClockPreload (`Clock`): clock is preloaded to a new value
ClockReadFault (`Clock`): reading the clock time has returned arbitrary value
ClockUpdate (`Clock`): clock is set to a new value
ClockWriteFault (`Clock`): clock time has been set to an arbitrary value
ConvergenceFailed (`ClockSync`): CV could not compute a new clock value
CrashStatusChange (`Clock`, `Lance`, `Network`): the module has changed its crash status (either has crashed, or has recovered)
DelayEstimation (`ClockSync`): new estimation of transmission delays
DeliverMessage (`Network`): the network has delivered a message to a CSA
DiscardRecvMessage (`Lance`): a message was discarded by the receiver CSA
DiscardSendMessage (`Lance`): a message was discarded by the sender CSA
DuplicateRecvMessage (`Lance`): a message was duplicated by the receiver CSA
DuplicateSendMessage (`Lance`): a message was duplicated by the sender CSA
EarlyMessage (`Network`): an early message is transmitted
GpsEvent (`Clock`): GPS time differs from the clock value
IgnoreSetting (`Clock`): write access to a clock register has failed (due to a crash)
IntervalEvent (`ClockSync`): contains either an input interval to the CV, or the result of the convergence function
LateMessage (`Network`): a late message is transmitted
MessageLost (`Network`): the network has discarded a message
RateEvent (`ClockSync`): contains either an input interval to the rate algorithm, or its result
ReceivedMessage (`ClockSync`): new message was received
ScrambleRecvMessage (`Lance`): a message was scrambled by the receiver CSA
ScrambleSendMessage (`Lance`): a message was scrambled by the sender CSA
SendMessage (`ClockSync`): a message is sent (passed to the network driver)
SendMessage (`Network`): a timely message is transmitted
Snapshot (`Clock`): data from a system snapshot
TimerEvent (`ClockSync`, `Clock`, `Lance`): clock duty timer ISR was called
TransmissionDelay (`ClockSync`): measured network transmission delay for a message
UserEvent (`ClockSync`): user defined event from the synchronization algorithm

The events are grouped into event classes. Three classes are available:

- SystemReport
- InterruptEvent
- FaultInjection

The mapping of events to event classes is shown in the following table:

SystemReport	InterruptEvent	FaultInjection
ChangeAssociation	GpsEvent	ClockReadFault
ClockPreload	ReceivedMessage	ClockWriteFault
ClockUpdate	SendMessage	CrashStatusChange
ConvergenceFailed	Snapshot	DiscardRecvMessage
DelayEstimation	TimerEvent	DiscardSendMessage
DeliverMessage		DuplicateRecvMessage
IntervalMessage		DuplicateSendMessage
RateMessage		EarlyMessage
TransmissionDelay		IgnoreSetting
UserEvent		LateMessage
		MessageLost
		ScrambleRecvMessage
		ScrambleSendMessage

References

[SynUTC] SynUTC-Homepage:
http://www.auto.tuwien.ac.at/Projects/SynUTC/

[Gri99] Günther Gridling: *Evaluation System for Clock Synchronization Algorithms*
Technical Report 183/1-89, Vienna University of Technology, Dept. of Automation, to appear
See the SynUTC-Homepage (Simulation Experiment)

[HSS98] Martin Horauer, Ulrich Schmid, Klaus Schossmaier: *NTI: A Network Time Interface M-Module for High-Accuracy Clock Synchronization*
6th International Workshop on Parallel and Distributed Real-Time Systems (WPDRTS), LNCS 1388, Orlando, USA, March 30 - April 3, 1998

[LM] M.C. Little, D.L. McCue: *Construction and Use of a Simulation Package in C++*
http://cxxsim.ncl.ac.uk/homepage.html

[Loy96] Dietmar Loy: *GPS-Linked High Accuracy NTP Time Processor for Distributed Fault-Tolerant Real-Time Systems*
PhD Thesis, Vienna University of Technology, Dept. of Computer Technology, April 1996

[Mar84] K. Marzullo: *Maintaining the time in a distributed system. An example of a loosely-coupled distributed service*
PhD Thesis, Stanford University, Dept. of Electrical Engineering, February 1984

[REX] REX lightweight process library: use anonymous ftp to dse.doc.ic.ac.uk, file /pub/rex/lwp.tar.gz

[Ri97] Gerda Richter: *Drivers for Real-Time Communication Coprocessors*
Diploma Thesis, Vienna University of Technology, Dept. of Automation, June 1997

[SS97] Ulrich Schmid, Klaus Schossmaier: *Interval-based Clock Synchronization*
Journal of Real-Time Systems 12 (2), March 1997

[Sch97] Ulrich Schmid: *Interval-based Clock Synchronization with Optimal Precision*
Technical Report 183/1-78, Vienna University of Technology, Dept. of Automation, December 1997

[SSHL97] Klaus Schossmaier, Ulrich Schmid, Martin Horauer, Dietmar Loy: *Specification and Implementation of the Universal Time Coordinated Synchronization Unit (UTCSU)*
Journal of Real-Time Systems 12 (3), May 1997

[Scho97] Klaus Schossmaier: *An Interval-based Framework for Clock Rate Synchronization*
16th ACM Symposium on Principles of Distributed Computing, St. Barbara, USA, August 21-24th, 1997, pp. 169-178

Related Literature

Daniel L. Palumbo: *The Derivation and Experimental Verification of Clock Synchronization Theory*
IEEE Transactions on Computers 43 (6), June 1994

Manfred J. Pflügl, Douglas M. Blough: *A New Model and Simulation Tool for Fault-Tolerant Clock Synchronization*
Proc. 22nd Annual Pittsburgh Conference on Modeling and Simulation 22 (3), May 1991, pp. 1604-1611

Anton V. Schedl: *Design and Simulation of Clock Synchronization in Distributed Systems*
PhD Thesis, Vienna University of Technology, Dept. of Technical Computer Science, April 1996

Installation

You can obtain the SimUTC archive (file `simutcXXX.tgz`, with `XXX` being the version number) from the SynUTC homepage (Simulation Experiment). Follow these steps to install the project:

tar -xz -f simutcXXX.tgz uncompresses the archive and installs the project files. The command will create a directory `SimUTC` (and several subdirectories) in your working directory and copy the files into it.

cd SimUTC changes into the new directory.

more install.txt lists the file which describes the rest of the installation procedure.