Simulation Environment for Clock Synchronization

project SimUTC

version 1.0

B. Weiss

June 18, 1997

Abstract

This thesis is concerned with the development of a simulation program for evaluating distributed interval-based clock synchronization algorithms. The program, which has been 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.

The following text can be seen as a "User's 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.

A floppy disk is included at the end of the thesis. It contains an installation guide, the program files, and documentation files describing the code.

Zusammenfassung

Diese Diplomarbeit beschäftigt sich mit der Entwicklung eines Simulationsprogrammes zur Auswertung verteilter Uhrensynchronisationsalgorithmen, die auf Intervallbasis arbeiten. Das Programm ist in C++ geschrieben und bedient sich der Methode der diskreten Simulation, um die Hardwareumgebung solcher Algorithmen zu simulieren. Der Benutzer kann das Verhalten dieser Umgebung beeinflussen und somit den zu testenden Algorithmus wohldefinierten Bedingungen unterwerfen. Das Verhalten des Algorithmus, das sich in der Güte der Synchronisation widerspiegelt, wird durch die Simulation beobachtet, die diese Daten dem Benutzer zur Verfügung stellt.

Diese Diplomarbeit ist eine Art erweitertes Benutzerhandbuch, das neben einer kurzen Einführung in die Problematik der Uhrensynchronisation im allgemeinen und in das für diese Applikation benötigte Wissen im speziellen hauptsächlich die Benutzerschnittstelle der Simulation beschreibt.

Auf der Innenseite des hinteren Umschlages befindet sich eine Diskette. Diese enthält neben Installationshinweisen die Programmdateien und deren Dokumentation.

Contents

1 Introduction **5**

2 Clock Synchronization **6**

3 System Overview **8**
3.1 Hardware-based Simulation 8
3.2 Pure Simulation 11

4 Simulation Modules **13**
4.1 Controller 13
4.2 Supervisor 15
4.3 ClockSync 16
4.4 Clock 19
4.5 Lance 20
4.6 Network 21

5 Changing the Algorithms **22**

6 Future Work **25**

A List of All Parameters **27**

B List of All Commands **28**

C List of All Events **29**

D Adding New Network Types **30**

E Interface to the Evaluation **32**
E.1 Message Format 32
E.2 Communication Protocol 33
E.3 Possible Message Types 34
E.3.1 Commonly Used Data 35
E.3.2 MsgCommand 36
E.3.3 MsgSetParameter, MsgGetParameter 40
E.3.4 MsgEvent 45
E.3.5 MsgReply 46
E.3.6 MsgModule 47
E.3.7 MsgByteOrder 47
E.3.8 MsgEnumSizes 47
E.4 Error Messages 48
E.5 Hardware-based Simulation 49

F Porting to Hardware **50**
F.1 Hardware Assumptions 50
F.2 Obsolete Files 52
F.3 New Files 52
F.4 Modifications to Files 52
F.5 Function (Re-)Implementations 53
F.6 Derived Classes 54
F.7 Tasks And Interrupts 55

References **60**

Further Literature **61**

Installation **62**

List of Figures

1 A Synchronization Period 7
2 Target System 9
3 Target System Areas 9
4 Simulation System Overview 11
5 Module Interaction 13
6 Function Calls to `ACClockSyncAlgorithm` 23
7 Message Format 33

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] [1] 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 [8], [10] on the Network Time Interface (NTI) hardware [3] built around our novel UTCSU (Universal Time Coordinated Synchronization Unit)-ASIC [5], 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.

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

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

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 thesis is concerned with part (1), 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. 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 my 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 change the synchronization algorithm.

Appendix D shows how to add new network types to the simulation, which needs some careful additions to the existing code.

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 Appendix E, which is dealing with the user (i.e., evaluation) interface of the simulation.

Appendix F contains instructions about 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 active part in the synchronization of one of their networks. The subnet in which a G-Node participates is called the *associated network*. 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.

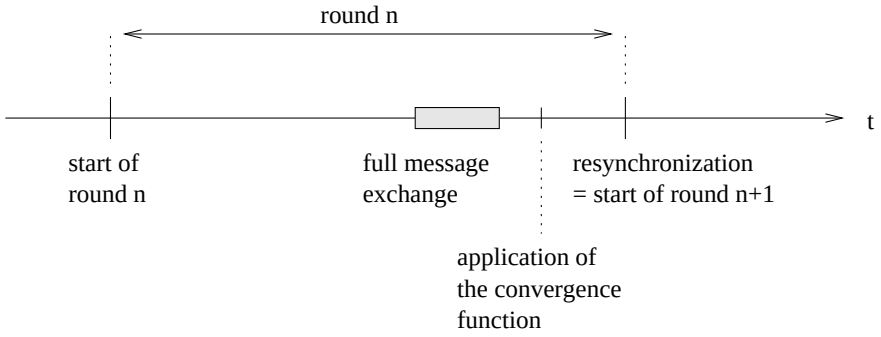

Figure 1: A Synchronization Period

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.

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 [10]).

This introduction is sufficient to understand the following description of the simulation system. See [3] for an excellent overview on interval-based clock synchronization. For a more detailed explanation of the synchronization algorithm, see [8].

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

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

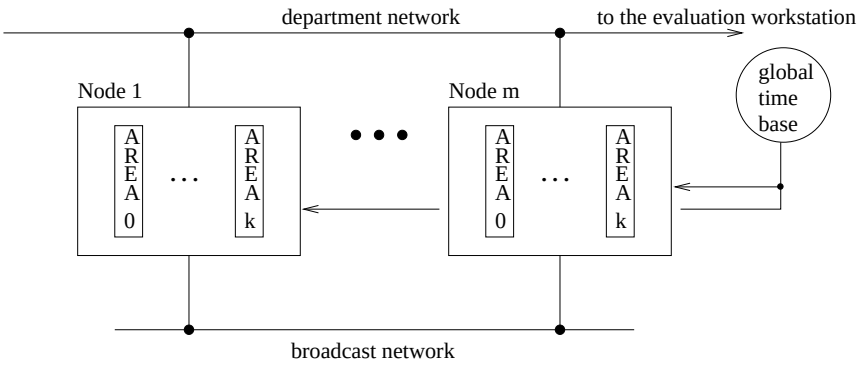

Figure 2: Target System

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.

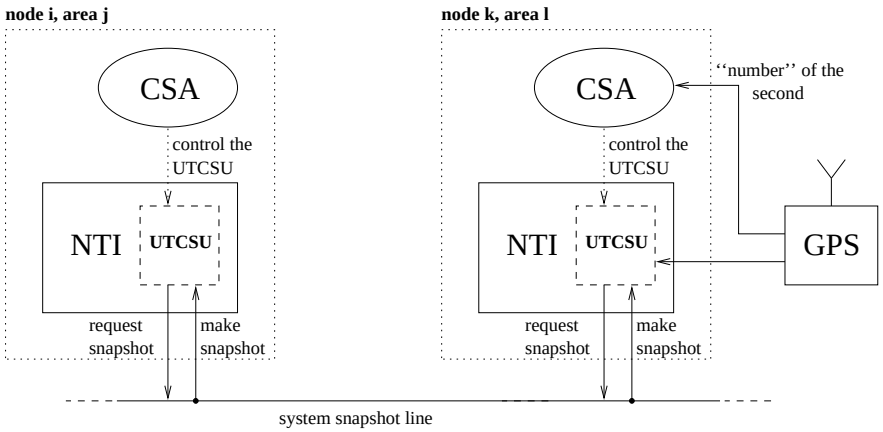

Figure 3: Target System Areas

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 [-8,-38] of the accuracies (only both at once, and to the same value).

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 [5] for an in-depth treatment of the features of the UTCSU, and [3] 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. [10]).

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 [4], which facilitates discrete-event simulation. The threads are provided by the REX lightweight process library [6]. 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.

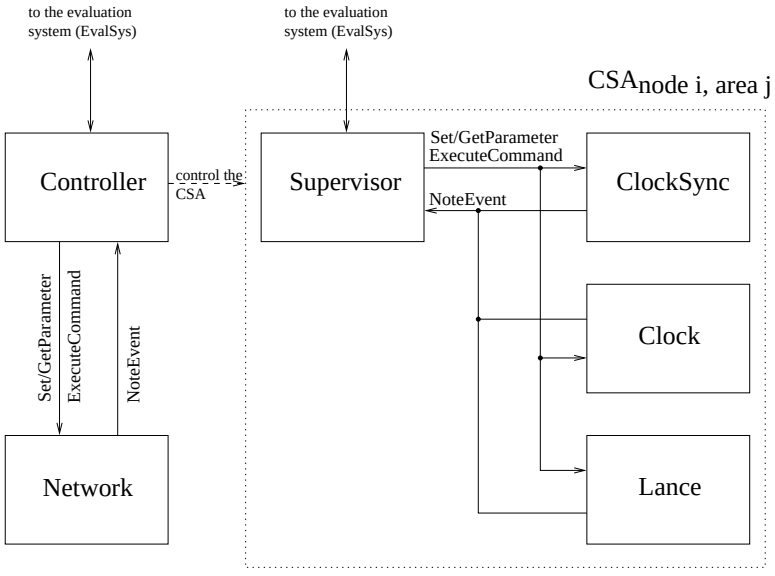

Figure 4: Simulation System Overview

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 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 `Network`s 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 Section E 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 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`

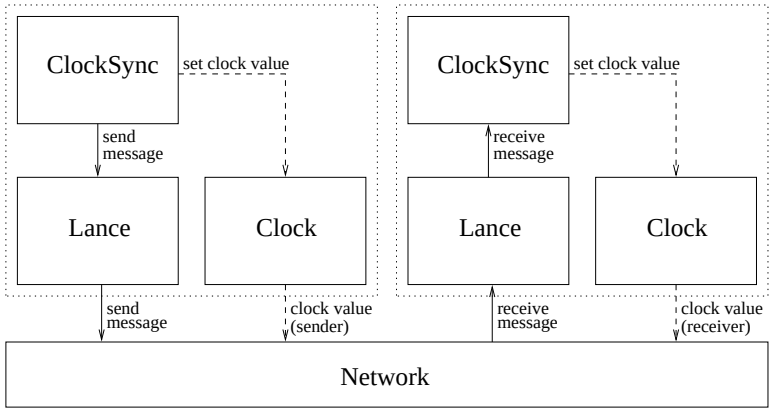

Figure 5: Module Interaction

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` 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 [2]). If it does not, or if you desire more knowledge of these elements, see Appendix E for a complete description of the interface to the evaluation.

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 I have explained above, the CSA will stop participating in the synchronization and ignore all messages from the other nodes. My current implementation will also discard any data (even delay- and rate measurements) it has 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*:

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[3]. The parameter is important for the synchronization algorithm.

RoundPeriod: controls the period[4] 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.

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

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.

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).

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.

[3]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.

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

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[5]. Note that independent access to these parameters is possible (even necessary), although they are grouped together in this description.

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.

AssociatedNetwork: sets the associated network of the node, i.e., the network to which the CSA synchronizes. As long as the associated network parameter is void, the CSA cannot take part in the synchronization.

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*:

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*:

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

ReceivedMessage: reports that a message was received.

DiscardMessage: is sent if a synchronization message was received but could not be used (because the module was suspended).

[5]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.

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

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

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

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 GPS, 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.

There are some *parameters* that may be accessed:

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.

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[6].

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 that information, so there is barely any difference between an imperfect hardware clock and a software clock with a non-zero drift.[7]

[6] Remember that in case of the UTCSU, the value corresponds to bits [-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.

[7] 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}.

RhoPlus, RhoMinus: sets the worst case (upper or lower) bound on the clock drift. Although they are listed together, the parameters are independent.

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.

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.

Only the following *command* is possible:

ResetModule: restores the initial state of the module.

And `Clock` may send these *events*:

ClockUpdate: is sent if the clock is set to a new value.

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

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

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 [7] for more information about the driver) and delivering received messages to the `ClockSync` module.

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 neither parameters nor events, only one *command* is available:

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

4.6 Network

The `Network` simulates the transmission of messages over a network. It accepts messages from the `Lances`, 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 D for instructions on what to do if another type of network is needed.

The network allows access to several *parameters*:

Delta, EpsilonPlus, EpsilonMinus: sets the message transmission characteristics (δ, $\epsilon^{\pm}$) for a pair of nodes. In contrast to module `ClockSync`, the user may create new entries. In fact, they *must* be created: 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.

DistributionTimely: sets the distribution function for choosing a transmission delay for a message that is transmitted between a pair of nodes. The delay is taken out of the interval $[\delta - \epsilon^{-}, \delta + \epsilon^{+}]$.

Currently, for asymmetrical intervals only uniform distribution is available, and you can additionally use an exponential distribution if all transmission delay intervals are symmetrical around δ. A normal distribution will be added in the future.

Lambda: sets the maximum broadcast latency λ, which is the time from the send request of the node to the point where the message is actually put on the network. The minimum latency is assumed to be zero.

DistributionLambda: sets the distribution function that is used for choosing the value of λ for a specific message. You can use uniform or exponential distribution.

Omega: sets the maximum broadcast operation delay ω, which 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. Again, the minimum delay is assumed to be zero.

DistributionOmega: sets the distribution function for choosing the ω of a message. Only uniform or exponential distribution is available.

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: 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, that 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.

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

The only network currently implemented is a broadcast network which does not send any events.

5 Changing the Algorithms

There are 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 `ACClockSynchronizationAlgorithm` in file `algbase.hpp`.

In order to obtain a working program, some simple algorithms were implemented:

- 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 using the formula given in the description of the generic algorithm in [8]. This interval is stored until the convergence function is called. The CV will go through the list of intervals and replace them with single values (the midpoint of the interval, not the original time value). 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.

- 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.

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

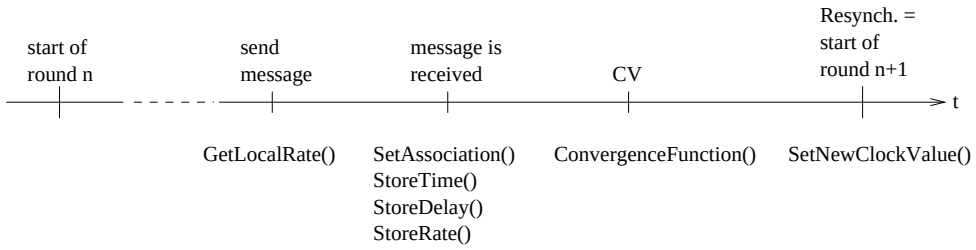

Figure 6: Function Calls to `ACClockSyncAlgorithm`

Fig. 6 shows which functions are called at which stages of the synchronization round. Please read the documentation `algbase.txt` 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`.

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 base 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 struct 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::Execute-Command()` 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::Report-Event()` 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.

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` [2] 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 replace the currently used class with your own. For that purpose, you first have to remove the file containing the old implementation from the list of project files (at least from the list of files that are linked together) and put your own file into that list. Module `ClockSync` obtains an object of the currently used algorithm class through calling function `CCsaTemplate::NewCSA()`. You have to define this function in your own implementation file (the declaration of the function can be found in file `algbase.hpp`). The function body is very simple: just create a new object of your class and return the pointer to it. After compiling your own file and relinking the project, your class will be used instead of the old one.

Note: 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 Future Work

This thesis describes version 1.0 of the project SimUTC, which enables the user to execute clock synchronization algorithms, but does not allow fault-injection, nor support repeated simulation runs.

The use of this version of the program lies mainly in providing the basic features a simulation program must have, like enabling the user to specify the behaviour of the environment (network, clock), to set the parameters of a clock synchronization algorithm, and to evaluate it under fault-free conditions. Since the algorithm is exchangeable, comparisons between different methods are possible.

Of course, this is only rudimentary support and not adequate for an in-depth evaluation of algorithms. Therefore, the next stage of the project must include probabilistic *fault-injection* capabilities. With this feature at hand, algorithms can be exposed to adverse conditions, making a long-term evaluation under user-definable conditions feasible.

The last and most difficult stage of the project is 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.

Whereas fault-injection has been long planned and its concepts are already clear, the replay mode will certainly be a big challenge for both the simulation program and the evaluation software.

A 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 transmission loss G_A (`Clock`)
- applied clock state corrections U during a rate round (`ClockSync`)
- associated network (`ClockSync`)
- broadcast latency λ (`Network`)
- distribution for choosing the actual broadcast latency in $(0, \lambda)$ (`Network`)
- broadcast operation delay ω (`Network`)
- distribution for choosing the broadcast operation delay in $(0, \omega)$ (`Network`)
- clock accuracy $\alpha^{\pm}$ (`Clock`)
- clock granularity G (`Clock`)
- clock time value (ms, ts) (`Clock`)
- computation delay compensation Γ (`ClockSync`)
- delay compensation $\Delta + \Lambda + \Omega$ (`ClockSync`)
- delay measurement cycles (`ClockSync`)
- deterministic δ and indeterministic $\epsilon^{\pm}$ part of the transmission delay (`Network`)
- distribution for choosing the transmission delay in $(\delta - \epsilon^{-}, \delta + \epsilon^{+})$ (`Network`)
- estimated bounds on the clock rate drifts $\theta^{\pm}$ (`ClockSync`)
- estimated transmission delay characteristics (δ, $\epsilon^{\pm}$) (`ClockSync`)
- event class report (`Controller`, `Supervisor`)
- event report (`Controller`, `Supervisor`)
- interface period (`Controller`, `Supervisor`)
- number of nodes n (`ClockSync`)
- number of faulty nodes f (`ClockSync`)
- actual rate drift ρ (`Clock`)
- bounds on the local rate drift $\rho^{\pm}$ (`Clock`)
- rate synchronization cycles (`ClockSync`)
- round period P (`ClockSync`)

There is one parameter that can only be read:

- timer granularity G_{CT} (`Controller`, `Supervisor`, `ClockSync`)

B List of All Commands

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

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

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

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

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

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

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

C List of All Events

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

- ClockUpdate (`Clock`): clock is set to a new value
- ConvergenceFailed (`ClockSync`) CV could not compute a new clock value
- DiscardMessage (`ClockSync`): `ClockSync` was suspended and had to ignore the message
- GpsEvent (`Clock`): GPS time differs from the clock value
- ReceivedMessage (`ClockSync`): new message was received
- SendMessage (`ClockSync`): a message is sent
- Snapshot (`Clock`): data from a system snapshot
- TimerEvent (`ClockSync`): clock timer ISR was called
- UserEvent (`ClockSync`): user defined event from the synchronization algorithm

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

- SystemReport
- InterruptEvent

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

event class	events
SystemReport	ClockUpdate ConvergenceFailed DiscardMessage UserEvent
InterruptEvent	GpsEvent ReceivedMessage SendMessage Snapshot TimerEvent

D Adding New Network Types

Adding new network types is quite simple, but unfortunately it requires some changes in the fixed part of the code and therefore recompilation of several project files.

The base class for all networks is the abstract class `ACNetwork`[8]. 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.txt` (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.

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

1. First, you have to enable `EvalSys` to request a network of your type. `EvalSys` specifies the network type using `enum ENetworkType` in `general.hpp`. You will have to add a new item to this enumeration for your network.

2. Now that your network type can be requested, you have to tell the `Controller` how to create a network of your type. For that, you will have to edit file `cont_sw.cpp`. In function `CController::CreateNewNetwork()`, add your network type to the list of types in the `switch`, and create the network at this place.

   If you need some additional information that are not provided by `struct SNetworkCreateCommand`, which is used for creating new networks, simply derive a struct and add your own members there. Likewise, you can make your own parameter, command, and event structs, which have to be derived 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`.

[8] See file `SimUTC/Sources/netbase.hpp` for the declaration, and file `SimUTC/Doc/Ascii/-Files/netbase.txt` for futher explanations of the class.

3. After you have made your additions to `general.hpp` and `cont_sw.cpp`, you will have to recompile the project files to take over the changes. If you link your own network implementation to the project, then the simulation will include your network.

4. In order to be able to use it from `EvalSys`, you will have to tell `EvalSys` what it has to do with the new network type. The evaluation system that is currently under development supports adding new networks during operation. Please confer to the documentation of `EvalSys` [2] for instructions on how to achieve this.

E Interface to the Evaluation

This chapter is intended mainly for the programmer of the evaluation system. It contains lists and descriptions of all parameters, commands and events that are passed between the simulation and the evaluation. Some "meta-messages", which are needed for establishing and maintaining the connection, are exchanged as well.

Every `Controller` and `Supervisor` module has its own stream socket to the evaluation. The message format is the same for all these sockets.

Apart from the normal communication sockets, the simulation may require a special datagram socket to send its error messages to. The following sections refer to the stream sockets and their communication protocol, see section E.4 for details on the datagram socket.

Please have a look at the simple text-based test "evaluation system" `evalsys.cpp` in directory `SimUTC/Sources/EvalSys` to see the socket communication at work. The most important thing is that `EvalSys` should create a well-known named socket[9] (immediately after startup) and `listen` to it. If a `Controller` is created, it will `connect` to that socket, thus creating its own connection socket. After the connection has been established (`EvalSys` has `accept`ed it), `EvalSys` can send commands to the `Controller` over this new socket, creating networks and CSAs. In the latter case, immediately after sending the command to the `Controller`, `EvalSys` should check its named socket again. It should `accept` the `connect` request of the `Supervisor`, which will again result in the creation of a connection socket solely for this `Supervisor`. Both `Controller` and `Supervisor` will send an identification message immediately after a successful connection, which will contain their module type.

E.1 Message Format

All messages consist of a header (one byte) and optionally a message identifier (four bytes) and the data, see Fig. 7. The data is parted into a leading *size* parameter (four bytes) and the data body. Parameter *size* denotes the size of the complete data block, including the four bytes of *size* itself.

The simulation will only send structures of type `SBase` (all structures can be found in file `structs.hpp`) or derived structures, so it is guaranteed that the first item in the data block is the size of the structure. When receiving data, the simulation always assumes that it can cast the data to struct `SBase`, and later on to one of the structures `SParameter` or `SCommand`.

[9] For pure simulation, the socket uses UNIX domain and it has the name `/tmp/SimUTC`. In hardware-based simulation, Internet domain will be used. The host and port names are still open.

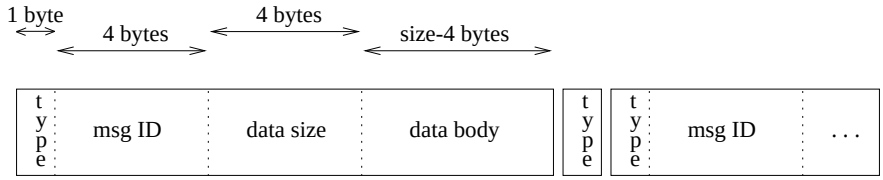

Figure 7: Message Format

Two messages differ from the above format. They only have the message type and then a data parameter of well-known size. See below for their descriptions.

E.2 Communication Protocol

The currently implemented protocol is quite simple, but differs slightly for the `Controller` and the `Supervisor`. Immediately after the modules have connected to the socket, the `Supervisor` sends one special message, and the `Controller` even three:

1. Both modules send their module type ("Controller" or "Supervisor") as first message after establishing the connection.

2. In case of the `Controller`, the second message is a well-known two-byte integer ($0x1234$) to inform `EvalSys` of the byte ordering on its system. If the byte ordering on the system used by the simulation is different from the order on the machine of `EvalSys` (i.e., if the evaluation reads $0x3412$), it is the task of `EvalSys` to convert every message to the simulation to the byte ordering it expects.

3. The third message sent by the `Controller` is a structure with the message format shown in Fig. 7 (i.e., type, identifier, and data structure), containing the sizes of all enumerations used in the communication between evaluation and simulation. Since the size of enumerations is not standardized, compilers may differ in their choice of enumeration sizes. Therefore, the evaluation must know the size of these enums so it can correctly assemble its own structures to match those used by the simulation. Note that the problem also includes the use of such basic types as `int`, so the sizes of these types are included into this message as well.

Afterwards, `Controller` and `Supervisor` show the same behaviour using the message format defined above. They periodically check the socket for messages (parameters, commands) from the evaluation and send any events collected since their last period to `EvalSys`.

If `EvalSys` sends a message (for example a command) to the simulation, this message should have a unique message identifier. Every command message

initiates some activity of the simulation, and the result of that activity is returned in a special reply (a structure derived from `SReply`) to the evaluation. Since there must be some way to tell the evaluation to which of its messages the reply belongs, the message identifier of the reply is the same as the identifier of the corresponding command message. The same holds for messages setting or reading parameters.

What happens if something goes wrong? If the simulation receives a message that is truncated or does not have the type expected? In that case, the message is discarded and a special message (Request For Resynchronization) is sent to notify the evaluation, which should respond with another dedicated message (Resynchronization). The simulation discards all data received from the time the error is detected and the resynchronization request is sent, up to the time the resynchronization message is received.

Of course, if the evaluation goes out of sync and requests a resynchronization, the simulation will respond with the resynchronization message. Please note that whereas the evaluation can find out which of the messages must have been lost (those to which the simulation has not replied) and can send them again, currently the simulation has no means to do such a roll-back. Future versions may change that.

E.3 Possible Message Types

As I have explained, the first item in a message is its type, which is one byte long. Although all message types are stored in an enumeration `EMessageType` (cf. file `params.hpp`), this is cast to `INT8` before it is sent on the socket. So it is guaranteed that the first item in every message is one byte long. The following list contains all currently used message types.

MsgByteOrder: sent from the simulation (`Controller` only) to the evaluation as the type of the second message. It is followed by a two byte unsigned (`INT16`) number.

MsgCommand: sent from the evaluation to the simulation to prompt a module to execute a command. The simulation will send a reply reporting whether the command could be executed successfully or not.

MsgEnumSizes: sent from the simulation to the evaluation as the type of the third message from the `Controller`. It contains the sizes of all data types with unspecified size used in the communication.

MsgEvent: sent from the simulation to the evaluation to report an event.

MsgGetParameter: sent from the evaluation to the simulation to read the value of a system parameter in a module. A reply will either contain the

value that was read, or tell the evaluation that the parameter could not be read.

MsgModule: sent from the simulation to the evaluation as the type of the first message. It is directly followed by a one byte unsigned (`INT8`) number specifying the module (type `EModule` is cast to `INT8` for that message). The module must be either `Controller` or `Supervisor`.

MsgReply: sent from the simulation to the evaluation in reply to a command or parameter message.

MsgResynchronizationByte: may be sent by both the simulation and the evaluation in response to a `MsgResynchronize`. The receiver will leave the resynchronization mode and again accept data from the socket.

MsgResynchronize: may be sent by both the simulation and the evaluation. The sender will go into resynchronization mode in which all data from the socket is discarded.

MsgSetParameter: sent from the evaluation to the simulation to change the value of a system parameter in a module. The simulation will reply if the parameter was set successfully or not.

E.3.1 Commonly Used Data

The following sections contain detailed descriptions of all possible messages, together with the destination module, the structure that is used as message body, and a list and description of all data that is needed.

Some data types will occur more than once and will have the same meaning in all their structures, so they are listed here.

networkID: the identifier of a network.
type: `SINT32` (i.e., a signed integral value with a size of four bytes)
range: $[0, \infty)$ [10]
comments: The ID is chosen by the simulation, which starts counting with 1 (although zero would be possible as well).

nodeID: the identifier of a node.
type: `SINT32`
range: [0, 4095]
comments: The identifier is chosen by `EvalSys` and the user.

[10] The symbol "∞" in this context signifies that the parameter may take every value from the lower bound up to the highest number that can be represented by the data type ($2^{31} - 1$ in this case).

area: the area on a specific node.

type: `int`

range: $[0, min(ASICCNT - 1, 15)]$ [11]

comments: The area is chosen by `EvalSys` and the user.

Apart from these three parameters, several enumerations are used. They can be found in the files `general.hpp`, `params.hpp` and `events.hpp`. All structures containing clock values are declared in `time.hpp`. The integral data types with fixed sizes (`INT32` etc.) are defined in `psos.h`. The boolean data type `BOOL` is declared in file `bool.hpp`.

Remember: all messages to the `Controller` and the `Network`(s) must be sent to the `Controller` socket. All messages to modules of the CSA must be sent to the appropriate `Supervisor` socket, which is dedicated to the CSA (so you will not need to specify the nodeID and area when communicating with CSAs).

E.3.2 MsgCommand

All commands will cause the simulation to send back a reply containing a boolean value (`True` or `False`). They return `True` if the command could be executed successfully, `False` if there was some problem. In the latter case, the command does not have any effect.

Some commands (those concerning the network) are only available in pure simulation. They may be sent in hardware-based simulation, but will always return `False` in that environment.

ConnectNode: connects a node to a network.

module: `Network`

structure: `SNetworkConnectCommand`, which allows you to specify the network to which you want to connect the node, plus the nodeID itself.

result: If the network exists, the node will be connected to it (note that there need not exist any CSAs on that node). The reply contains `True` if the network exists and the command could be executed successfully (for the implemented broadcast network, the command is always successful), otherwise `False`.

comments: The interpretation of "connecting the node" may vary among network types. The implemented network will accept send requests from the connected node, and it will transmit messages to this node. Other network implementations may only enter that node into their list of sender nodes, or just allow the node to receive messages.

[11] "ASICCNT" is a constant of the network driver (see [7]) and denotes the number of areas per node. Most likely, it will equal four. The simulation can handle up to 16 areas.

CreateApplication: creates a new CSA.

module: `Controller`

structure: `SApplCreateCommand`, so you have to specify the nodeID and area of the CSA you want to create. Apart from that, you need to request a node type (P- or S-Node). In case of P-Nodes, a GPS receiver will be simulated.

result: The command will return `True` if all data parameters are within their ranges and if no CSA with the given pair $\langle nodeID,\ area \rangle$ exists. In that case, the CSA will be created. Otherwise, `False` is returned.

comments: Note that the "node" type really is a property of the area, so two CSAs with the same nodeID may have different node types.

You may have noticed that there are no G-Nodes. Every node has the capability to be a G-Node, you just need to connect the node to more than one network. Do not forget to set the associated network for every CSA.

If real hardware is used, the simulation must be started on every node. So each node will have its own `Controller` and the nodeID is fixed for the life of the simulation. Nevertheless, you will have to specify a nodeID, which should be the same for all such commands to the `Controller` of a certain node.

After creation, the CSA is suspended. The `Supervisor` of the CSA will connect to your named socket immediately after it is created. You should configure the CSA to your liking before you resume the application.

CreateNetwork: creates a new network.

module: `Controller`

structure: `SNetworkCreateCommand`, so you have to specify the network type (right now, only a network of type `Broadcast` is available).

result: If the network type is known, a new network will be created and a unique identifier (networkID) will be appointed to it. Since you will need that identifier for all further interaction with the network, it is returned to you in a reply of type `ReplyDouble` (the `SINT32` is cast to `double` for that). This message will be followed by the normal boolean reply containing `True`. If the network could not be created for any reason (because the type was not known or not supported), only the boolean reply will be sent, and that message will contain the value `False`.

DeleteApplication: deletes an existing CSA.

module: `Controller`

structure: `SApplCommand`, so you have to specify the nodeID and area of the CSA.

result: If the CSA exists, it will be deleted and `True` is returned. Otherwise, the command will return `False`.

DeleteNetwork: deletes an existing network.
module: `Controller`
structure: `SNetworkCommand`, which contains the networkID of the network that is to be deleted.
result: If a network with that ID exists, it will be deleted and `True` is returned. Else `False` is returned.

DisableSnapshot: disables system snapshots (default after startup).
module: `Supervisor`
structure: `SCommand`
result: The command will always return `True`.
comments: If the feature was already disabled, the command has no effect.

DisconnectNode: disconnects a node from the network.
module: `Network`
structure: `SNetworkConnectCommand`, so you have to specify the network and the ID of the node you want to disconnect.
result: The command returns `True` if the network exists (for the implemented network, disconnecting a node is always successful, even if the node was not connected), `False` if the networkID is not in use.
comments: Like command `ConnectNode`, interpretation of the command depends on the network type. The implemented broadcast network will refuse to transmit any messages from that node, nor send any messages to it.

EnableSnapshot: enables system snapshots.
module: `Supervisor`
structure: `SCommand`
result: The command will always return `True`.
comments: If the feature was already enabled, the command has no effect.

KillSimulation: deletes the simulation, exits the program.
module: `Controller`
structure: `SCommand`
result: The command will always return `True`.
comments: It deletes all networks and CSAs that were created and then stops the program.

ResetModule: that command may be sent to every module in the simulation, causing that module to restore its initial state. You have to distinguish between two situations:

- *module:* `Network`
  structure: `SNetworkCommand`, so you have to specify the networkID.

result: The command returns `True` if the network with the given ID exists, otherwise `False`.

- *module:* all other modules
  structure: `SCommand`
  result: The command will always return `True`.

ResetSimulation: resets all simulation objects.
module: `Controller`
structure: `SCommand`
result: It returns `True` if at least one CSA or network exists, `False` if no simulation modules exist.
comments: The command resets all CSAs and networks that were created.

If there are any events still to be sent to `EvalSys`, they are discarded.

ResumeApplication: resumes a CSA.
module: `Controller`
structure: `SApplCommand`, so you need the nodeID and area of the CSA.
result: If the CSA exists, it will be resumed and `True` is returned. Otherwise, the command will return `False`.

ResumeSimulation: resumes all CSAs.
module: `Controller`
structure: `SCommand`
result: If at least one CSA exists, all CSAs are resumed (that has no effect on CSAs that were not suspended) and `True` is returned. If no CSAs exist, `False` is returned.

SuspendApplication: suspends a CSA.
module: `Controller`
structure: `SApplCommand`, so you need the nodeID and area of the CSA.
result: If the CSA exists, it will be suspended and `True` is returned. Otherwise, the command will return `False`.

SuspendSimulation: suspends all CSAs.
module: `Controller`
structure: `SCommand`
result: If there are CSAs to suspend (i.e., if at least one CSA exists), they are all suspended and `True` is returned. If no CSAs exist, `False` is returned.

UserCommand: a command to the user-definable part of the CSA.
module: `ClockSync`
structure: derived from `SCommand`
result: The command has user defined data and an unknown result. In any case, the result of the command must be a boolean value.

E.3.3 MsgSetParameter, MsgGetParameter

This is a sorted list of all parameter types, together with the structure that is used for setting/reading the value and with a list of all data needed to access the parameter. The type and range of parameters are given, as well as their default values for the simulation. Some parameters have to obey certain conditions, which are pointed out as well. `EvalSys` should take care that parameters stay within their allowed range and meet all their conditions.

Note that if there is no range specified, the full range of the data type can be used.

Accuracy: accuracy of a clock
module: `Clock`
structure: `SSingleParameter`
type: `INT32`
default: the initial accuracies are 0
comments: The clock has an upper and lower accuracy, and this parameter will access both of them.

For setting the accuracy, the `INT32` contains a signed 31 bit accuracy value, where the lsb of the value is 2^{-38} seconds. Both accuracies will be set to the value handed over.

When reading the parameter, both accuracies are read into the `INT32`: the upper two bytes hold the lower accuracy α^- (the lsb of that 16 bit value is 2^{-23} seconds), the lower two bytes store the upper accuracy α^+ (same resolution). Negative accuracies are read as zero.

AccuracyTransmissionLoss: granularity of the clock accuracy
module: `Clock`
structure: `SSingleParameter`
type: `double`, specified in seconds
range: $(0, \infty)$
default: 2^{-23} seconds (the lsb of the transmitted accuracy)

AssociatedNetwork: network in which a CSA participates
module: `ClockSync`
structure: `SSingleParameter`
type: `SINT32`
default: no associated network
comments: After creation, a CSA does not have any associated network and thus will not be able to take part in the synchronization. So you have to specify an associated network for every CSA you create.

ClockDrift: drift of a clock
module: `Clock`

structure: `SSingleParameter`
type: `double`, specified in seconds/second
range: (-1,∞)
default: 0
comments: The parameter is only available in pure simulation.

ClockGranularity: granularity of a clock
module: `Clock`
structure: `SSingleParameter`
type: `double`, specified in seconds
range: (0, ∞)
default: 2^{-24} seconds (the lsb of the clock timestamp)

ClockValue: value of a clock
module: `Clock`
structure: `STimeParameter`
type: `INT32` for both macrostamp and timestamp
default: the initial clock value is 0
comments: The lsb of the macrostamp is 2^8 seconds, and the highest value for the macrostamp is $2^{32} - 1$ seconds. The uppermost byte of the macrostamp is ignored, `EvalSys` should set it to zero.

The lsb of the timestamp is 2^{-24} seconds, and all four bytes of the timestamp can be used.

ComputationDelayCompensation: estimate of the maximum execution time of the CV
module: `ClockSync`
structure: `SSingleParameter`
type: `double`, specified in seconds
range: [0, $P-(\Delta+\Lambda+\Omega)$], where P denotes the `RoundPeriod`, and $\Delta+\Lambda+\Omega$ is the `DelayCompensation`
default: 0
comments: `EvalSys` has to ensure that the parameter is a multiple of the clock timer granularity G_{CT}.

DelayCompensation: waiting time for message arrivals
module: `ClockSync`
structure: `SSingleParameter`
type: `double`, specified in seconds
range: [0, $P - \Gamma$], where P denotes the `RoundPeriod`, and Γ is the `ComputationDelayCompensation`
default: 1 second (1% of the default value of P)
comments: Like `ComputationDelayCompensation`, this parameter must be a multiple of G_{CT}.

DelayMeasurementCycles: period of delay measurement
module: `ClockSync`
structure: `SSingleParameter`
type: `INT32`, specified in rounds
default: 0 (disabled)
comments: If the parameter is set to k, every k-th FME-message contains a request for delay measurement, starting with the message sent in the next round. Setting $k = 0$ disables this feature, which is the default.

Delta: there are two possibilities for this parameter:

- *module:* `ClockSync`
  structure: `STransmissionData`, so you have to specify the networkID of the transmitting network and the nodeID of the sender node.
  default: none
  comments: You can only modify existing transmission entries.
- *module:* `Network`
  structure: `SNetworkData`, so you need the networkID of the network module and the nodeIDs of the sender and receiver node.
  default: $sender \neq receiver$: none, $sender = receiver$: 0
  comments: New entries can be created.

type: `double`, specified in seconds
range: $[0, \infty)$
comments: Parameter δ is part of the transmission delay characteristics $(\delta, \epsilon^+, \epsilon^-)$ of a network. If such a triple is created anew (because one parameter was set for the first time), the other two are set to zero as default.

DistributionLambda: distribution function
module: `Network`
structure: `SSingleNetworkParameter`, so you need to specify the networkID.
type: `EDistribution`
default: Uniform
comments: Currently, you should only use type "Uniform" or type "Exponential". Type "Normal" may be chosen, but returns a uniform value.

DistributionOmega: Same as `DistributionLambda`.

DistributionTimely: Same as `DistributionLambda`.
comments: Currently, you should only use type "Uniform" for asymmetrical intervals. You could use type "Exponential" if you have symmetrical transmission delay intervals.

EpsilonMinus: see `Delta`.
range: $[0, \delta]$

EpsilonPlus: see `Delta`.
range: $[0, \infty)$

EventClassReport: to control filtering of complete event classes
modules: `Controller`, `Supervisor`
structure: `SEventReport`, which offers functions to set a single class or a whole mask to True or False.
default: all event classes are disabled
comments: See Appendix C for the mapping of events to event classes.

EventReport: to control filtering of single events
modules: `Controller`, `Supervisor`
structure: `SEventReport`, which offers functions to set a single event or a whole mask to True or False.
default: all events are disabled

InterfacePeriod: response time of communication with `EvalSys`
modules: `Controller`, `Supervisor`
structure: `SSingleParameter`
type: `double`, specified in seconds
range: $(0, \infty)$
default: 1
comments: Since a pSOS^{+m} timer is used for achieving the period, the parameter should be a multiple of the pSOS^{+m} tick granularity.

Lambda: broadcast latency of a network
module: `Network`
structure: `SSingleNetworkParameter`, so you have to specify the networkID.
type: `double`, specified in seconds
range: $[0, \infty)$
default: 0

NumberOfFaultyNodes: (maximum) number of faulty nodes (=CSAs) in the system
module: `ClockSync`
structure: `SSingleParameter`
type: `INT32`
range: $[0, n]$, where n is the number of nodes in the system
default: 0
comments: Most likely, this parameter will be much less than n, e.g., it may be $f < n/2$ where f denotes the number of faulty nodes.

NumberOfNodes: number of nodes (=CSAs) in the system
module: `ClockSync`
structure: `SSingleParameter`
type: `INT32`
range: $[1, \infty)$
default: 1

Omega: See `Lambda`.

RateSynchronizationCycles: See `DelayMeasurementCycles`.
comments: If the parameter is set to k, every k-th FME-message contains additional rate information.

RhoMinus: lower bound on the clock rate
module: `Clock`
structure: `SSingleParameter`
type: `double`, specified in seconds/second
range: $[0, 1)$
default: 10^{-3}

RhoPlus: like `RhoMinus`.
range: $[0, \infty)$

RoundPeriod: period of the synchronization round
module: `ClockSync`
structure: `SSingleParameter`
type: `double`, specified in seconds
range: $[0, \infty)$
default: 100
comments: The value of the parameter has to be a multiple of G_{CT}.

ThetaMinus: computed lower bound of the clock rate
module: `ClockSync`
structure: `SSingleParameter`
type: `double`, specified in seconds/second
range: $[0, 1)$

ThetaPlus: like `ThetaMinus`.
range: $[0, \infty)$

TimerGranularity: granularity of the timer used for achieving a period
modules: `Controller`, `Supervisor`, `ClockSync`
structure: `SSingleParameter`
type: `double`, specified in seconds
comments: The parameter is read-only.

For the modules using the pSOS^{+m} timer (`Controller` and `Supervisor`), the value of the parameter is 10^{-2} seconds (one tick).

For `ClockSync`, which uses the UTCSU, its value is 2^{-16} seconds (the resolution of the clock timer).

U: accumulated state adjustments during a round
module: `ClockSync`
structure: `SSingleParameter`
type: `double`, specified in seconds
range: $[0, \infty)$
default: 0

UserParameter: user-definable parameter
module: `ClockSync`
structure: derived from `SParameter`
result: The parameter holds user defined data and results.

E.3.4 MsgEvent

Event messages are sent from the simulation to the evaluation to report all kind of interesting things that happen during the course of executing a synchronization algorithm. The simulation does not expect any reply.

ClockUpdate: clock is set to a new value
module: `Clock`
structure: `SClockEvent`, which apart from information to identify the CSA (nodeID, area) contains the global time of the event report (a `double` value, specified in seconds), the current clock value, and the new clock value to which the clock will be set.
event class: `SystemReport`

ConvergenceFailed: the convergence function could not compute a new clock value
module: `ClockSync`
structure: `SApplicationEvent`, which contains the nodeID and area of the CSA, plus the global time and local clock value at the time of the event report.
event class: `SystemReport`

DiscardMessage: message was discarded
module: `ClockSync`
structure: `SApplicationEvent`
event class: `SystemReport`

GpsEvent: GPS time differs from clock value
module: `Clock`
structure: `SClockEvent`, where the new clock value is obtained from the GPS receiver.
event class: `InterruptEvent`

ReceivedMessage: message was received
module: `ClockSync`
structure: `SApplicationEvent`
event class: `InterruptEvent`

SendMessage: message was sent
module: `ClockSync`
structure: `SApplicationEvent`
event class: `InterruptEvent`

Snapshot: snapshot was taken
module: `Clock`
structure: `SClockEvent`, where the `newTime` entry contains the clock value at the time of the snapshot.
event class: `InterruptEvent`

TimerMessage: clock timer has produced an interrupt
module: `ClockSync`
structure: `SApplicationEvent`
event class: `InterruptEvent`

UserEvent: user-definable event
module: `ClockSync`
structure: derived from `SApplicationEvent`
event class: `SystemReport`

E.3.5 MsgReply

Replies are messages from the simulation that are sent in response to messages from `EvalSys`. The simulation does not expect any reply.

ReplyBoolean: this is the reply to all `MsgCommand` and `MsgSetParameter` messages from `EvalSys`. It is also used in response to a `MsgGetParameter` that has failed.
structure: `SBooleanReply`.

ReplyDouble: is used to send the networkID of a newly created network to `EvalSys`. It is only sent in reply to a command of type `CreateNewNetwork`, given that it has succeeded.
structure: `SDoubleReply`

ReplyStructure: is sent in response to a successful `MsgGetParameter` message. If `EvalSys` wishes to read a parameter, it should send the appropriate structure for accessing the parameter, but not set the value. The module storing the parameter will set the value and send a reply of type `ReplyStructure` which consists of structure `SReply` immediately followed by the parameter structure. The `size` member of structure `SReply` will be modified to enclose both its own structure and the parameter structure. When reading the reply, do not forget that you are really reading two structures and split them.

E.3.6 MsgModule

This is the first message of any interface module (`Controller` and `Supervisor`) that is newly created. After `EvalSys` has accepted a connection, it should wait for this message to determine (or verify) the module type. Note that this is not really necessary in pure simulation, since here the first socket connection must be the `Controller`, and all other connections should be from `Supervisor`s. In hardware-based simulation, however, several `Controller` modules can exist (they are replicated for every node), and the order of connection requests from the `Controller`s and `Supervisor`s is not guaranteed anymore.

The header is immediately followed by an `INT8` containing the type of the module. This parameter is of type `EModule`, but cast to `INT8` to guarantee a data size of 1 byte.

E.3.7 MsgByteOrder

This is the second message of any `Controller` that is newly created. The header is followed by an `INT16` (no message ID, no size). The `INT16` should read $0x1234$ if the byte ordering is the same on both systems. If $0x3412$ is received, `EvalSys` has to convert every message to the format used by the simulation. Consequently, `EvalSys` should also convert every message from the simulation into its own format. Any differences in byte ordering must be completely transparent to the simulation.

E.3.8 MsgEnumSizes

The third message from a newly created `Controller` contains the sizes of all enumerations used in the communication structures derived from `SBase`. For that purpose, structure `SEnumSizes` is used. This message does already include a message ID, so it can be read like any other message.

E.4 Error Messages

For debugging purposes, the simulation can be configured to print all kinds of error messages. Since the target environment of the simulation may not include an output device, the simulation can redirect its output to a datagram socket. So after startup, `EvalSys` should not only create the stream socket to which the application modules connect, but also a datagram socket for error messages[12].

`EvalSys` should poll the socket and receive the error messages. These messages are simple strings, and the evaluation should print them (followed by a newline) to the screen and/or to some error log.

If `EvalSys` does not empty the socket fast enough, the simulation may experience send errors if the socket buffer is full. In this case, the simulation will repeat the send operation for a limited number of times, and if it does not succeed, it will give up on the socket and try to print the message to the local output device instead. In pure simulation, this is no major problem, but in hardware-based simulation, error messages can get lost if no output device is available. Hence, `EvalSys` should try to poll the socket fast enough.

Note that the problem only occurs if there are many errors at once. You may never experience such a situation in pure simulation, and nevertheless encounter it in hardware-based simulation. I advise you to test the quality of error handling. For this purpose, a small test program has been added. You can find it in directory `SimUTC/Sources/Socktest`. Call `make` to create a datagram client. Start your evaluation to provide the datagram socket. Then start the client with `dclient` (preferably in its own shell). The evaluation should receive a couple of messages (`report: test message`) from the client. Check if there are any socket error reports among these messages. Also check if `dclient` prints any test or error messages. If the evaluation receives any socket error reports, then at times the socket was full and the client had to try to send a message more than once. If the client prints any messages, this means that it repeatedly could not send a message to the socket (constant `MaxTrials` in `constant.hpp` controls the number of attempts to send a message) and has written it to the local output instead. In this case, you are too slow to handle a message burst and should either try to poll the socket more frequently, or get the programmer of the simulation to set `MaxTrials` to a higher value.

If you receive no socket error messages, repeat the call to the client several times, or edit the source code `dclient.cpp` and set the constant `NumberOfMessages` to a higher value (use a multiple of `MaxErrorMessages`) to get more messages. If several runs do not produce any errors, you can assume that you empty the datagram socket sufficiently fast.

[12] In pure simulation, UNIX domain is used, and the name of the socket should be `/tmp/SimError`. Hardware-based simulation requires Internet domain, where the host and port names are still open.

E.5 Hardware-based Simulation

This section sums up the differences between pure and hardware-based simulation. These items were already mentioned in the previous sections, but are assembled in the following table.

	pure simulation	hardware-based sim.
number of `Controller`s	1	$\leq$ #nodes (4)
simulated networks	yes	no
clock drift changeable	yes	no
stream socket	address family (AF): UNIX name: /tmp/SimUTC	AF: INET host, port: still open
datagram socket	AF: UNIX name: /tmp/SimError	AF: INET host, port: still open

Since `EvalSys` does not know the type of the simulation in advance, you should create both the UNIX and Internet sockets and use non-blocking I/O for the stream sockets. Then check both stream sockets until you get a connection request from one of them. If the `Controller` connects to the UNIX socket, you are working with pure simulation. If it connects to the Internet socket, hardware-based simulation is used.

After you have established the type of the simulation, you can close the sockets that are not needed.

F Porting to Hardware

Porting to the hardware means adapting the simulation software for execution on the target system working with a real network, the LANCE driver [7], the NTI [3] and the UTCSU [5]. The operating system will be pSOS^{+m}. Porting the pure simulation should be a rather straightforward process of (re-)implementing some functions, removing certain header and object files that are only needed in pure simulation, and deriving a few classes. The source files can be found in directory `SimUTC/Sources`.

The hardware-based simulation differs in some aspects from the pure simulation. Apart from the fact that it works with the target, which may need hardware and software configuration before you can start the simulation (have a look at the appropriate manuals, which should be available as soon as the hardware environment is established), the major difference is that you will have to start the simulation on each target (node). Therefore, each target executes its own simulation (with its own `Controller`), which are co-ordinated by `EvalSys`.

F.1 Hardware Assumptions

The simulation software poses some general assumptions and requirements to the hardware. If any of them are not met, re-engineering of the fixed part of the code becomes necessary.

- The operating system is pSOS^{+m}.
- Socket communication using TCP/IP stream and datagram sockets in Internet domain is supported.
- A global time base available to all CSAs on all nodes does exist. The time base should have a granularity of about 1 μs or less. It does not have to be accurate w.r.t. external time, since it is only used for event ordering. Obtaining a timestamp should be fast.
- The register layout (and behaviour) of a hardware clock is the same as that of the UTCSU described in [5]. Each clock is memory-mapped into the address space of the simulation.
- If the clock generates an interrupt, the type of the interrupt (INTA, INTN, and/or INTT) is coded into the lowest 3 bits of the base vector of the clock according to the following table (combinations of the interrupt types are possible):

interrupt type	value ORed to the base vector
INTA	0x01
INTT	0x02
INTN	0x04

- The APPDUTY output pins of all UTCSUs are connected to the HWSNAP input pins of all UTCSUs. This is necessary for making system snapshots, where all clocks sample their values simultaneously.

- Interrupts of the GPS receivers, indicating that the time information (the "number" of the second) is available, can be enabled/disabled independently from each other. After starting the program, these interrupts are disabled.

- If enabled, the GPS receiver generates an interrupt when its time information is available. This time can be read and converted into the format of a clock value (macrostamp, timestamp).

- If two or more GPS receivers share an interrupt vector, they must be connected to the same UTCSU, and it must be possible to find out which particular receiver has generated the interrupt.

- If a UTCSU has GPS receivers connected to it, GPU1 must be used, and all other receivers of this UTCSU must have the same interrupt vector.

  Note that the condition is only needed in this version of SimUTC, and will not be required by future versions.

- The interface to the network is the LANCE driver, whose CI functions must have the behaviour described in [7].

- Every message that is sent must be timestamped with the clock value (macrostamp, timestamp, accuracy) of the sender just when it is put on the network. At the receiver, the message should be timestamped again in the moment it is taken from the network. The send timestamp (plus its area) must be included as the leading items in the message that is announced to the receiver. So the message buffer passed by `ci_announce()` should contain a `struct {INT32 area; ti_acx time;}` as its first entry, and the data transmitted by the sender should follow it.

  This strict order, and the types of the `area` and `ts`, `ms`, `acc` parameters of `ti_acx` (all INT32), are only required by this version of the simulation. Future versions will allow independent and arbitrary placement of these parameters in the receive message buffer, and support configurable types.

F.2 Obsolete Files

Several files can be removed from the project (i.e., do not link them to the project and do not use them in your own files in any way), since they are not needed for execution on hardware. These files are

> ci.cpp, clockirq.cpp, clockirq.hpp, cloctemp.hpp, clock_sw.cpp, clock_sw.hpp, cont_sw.cpp, cont_sw.hpp, duplist.cpp, duplist.hpp, evalglob.hpp evalsys.cpp, evalsys2.cpp (that may not be there anyway), evaltemp.hpp, gps_sw.cpp, gps_sw.hpp, idlist.cpp, idlist.hpp, impl_sw.cpp, intproc.cpp, intproc.hpp, irqtemp.hpp, itoa.cpp, main.cpp, msgtemp.hpp, lantemp.hpp, netacc.cpp, netacc.hpp, netbase.cpp, netbase.hpp, netgen.cpp, netgen.hpp, netproc.cpp, netproc.hpp, nettemp.hpp, network.cpp, network.hpp, psos.cpp, psos.hpp, sim_gen.cpp, sim_gen.hpp, sim_glob.cpp, sim_glob.hpp, tester.cpp, tester.hpp, unixdefi.hpp (if it is there; do not confuse it with unixdefines.hpp!)

A couple of header files have to be removed, because in hardware they do already exist:

> ci.h (use the global file of the LANCE driver), error.h (LANCE driver), nti.h (use global NTI headers), psos.h (LANCE driver)

F.3 New Files

The following files should be added to the project, see below for a description of their contents:

> clock_hw.cpp, clock_hw.hpp, cont_hw.cpp, cont_hw.hpp, gps_hw.cpp, gps_hw.hpp, impl_hw.cpp

F.4 Modifications to Files

Some files will have to be modified:

unixdefines.hpp: Three defines switch between operating systems: `LINUX_SIM` is used for pure simulation under Linux (using the simulation package C++SIM), `DOS_SIM` is used for development under Win95, and `PSOS_SIM` is used for hardware-based simulation under pSOS^{+m}. So you need to set the defines like this:

```
//#define DOS_SIM
//#define LINUX_SIM
#define PSOS_SIM
```

F.5 Function (Re-)Implementations

All functions listed below should be implemented in the file impl_hw.cpp. For comparison, see impl_sw.cpp, which contains their implementations in software, with the exception of the CI functions which are not needed in pure simulation. The file name in brackets is the header file containing the function prototype[13].

ci_announce(): [ci.hpp]

The network driver assumes the existence of this function. It is called whenever a message is received on the node. You should go through the `listOfSupervisors` in `globals.hpp`, get the `Lance`s from the `Supervisors`, and call their `Announce()` function. This function assumes that the buffer is contiguous, so you should call `ci_receive()` if necessary. Since the `Lance` does not return the buffers, `ci_announce()` should exit with the return value `IMMEDIATE_RELEASE` (causing the CI driver to release the buffers).

ci_announce_error(): [ci.hpp]

Again, the network driver assumes the existence of this function. It is called if a message could not be transmitted. Since my `CLance` class does not support error handling, I do not care what you do with the data. I suggest calling the error report macro `Report()` [`error.hpp`] with an error message and discarding the packet (i.e., returning the buffers).

By the way, if you ever feel the need to assert, check or report something, please use one of the macros in `error.hpp`. Do not use any own functions for output.

GetClockIntvec(): [interupt.hpp]

`GetClockIntvec()` should return the NTI interrupt (base) vector of the UTCSU on the specified area. The nodeID parameter can be ignored.

GetGpsIntvec(): [interupt.hpp]

`GetGpsIntvec()` should return the NTI interrupt vector of the GPS receiver connected to the given GPU of the UTCSU on the specified area. The nodeID parameter can be ignored.

Note that the simulation requires that every area with GPS has its own GPS vector(s). If vectors are shared between two GPS receivers, both must be connected to the same UTCSU.

Note: the current version does not allow more than one GPS interrupt vector per UTCSU. If you connect more than one receiver to a UTCSU,

[13] I usually omit all parameters when I am refering to functions, so you should always have a look at the header file if you need to know the correct prototype of a function.

you should always connect one to GPU1, and all others should have the same vector as the receiver on GPU1.

GetIntLevel(): [interupt.hpp]

Return the interrupt level of the UTCSU, GPS and LANCE. Take care when setting the interrupt levels: all components should have the same level. If you use different levels, race conditions may occur.

GetGlobalTime(): [general.hpp]

There should be some global time base available to all nodes. How to generate such a time base is still open. The currently favoured approach is to feed an oscillator output to one of the application inputs on all UTCSUs. So whenever a new tick is generated, the current clock value is sampled into the application registers. Between two ticks, which should be close together, the clock is assumed to have approx. ideal rate, so you can compute the global time from the number of the tick, plus the difference of the current clock value from that sampled at the occurance of the last tick.

Still unsolved is the problem of conveying the tick number to all nodes, since they need to know the number of an oscillator tick for consistency reasons. So this solution may not be feasible.

F.6 Derived Classes

The following classes should be derived and implemented. They interact with the hardware.

CHardwareClock: [clock_hw.cpp/hpp]

This is the hardware version of the access class to the UTCSU, derived from `CClock` in file `clockbas.hpp`. Most likely, you will not need to overload any functions of the base class and can use the existing implementation in file `clock_hw.cpp`.

Note: if you do not use that file, remember that there is a template[14] for obtaining an instance of the clock. You have to implement the function `CClockTemplate::NewClock()` (see my version of `clock_hw.cpp`).

CHardwareController: [cont_hw.cpp/hpp]

The hardware version of the controller, derived from `ACController` in file `contbase.hpp`. It is not so much concerned with hardware but with the operating system: when using pSOS^{+m}, every application is a task and has to be created/started using function `StartApplication()`, and is deleted

[14]Not in the sense of a C++ template, but a wrapper class that will create an object of the currently used clock implementation (hardware or software clock).

with function `StopApplication()`. The `Controller` is a task as well (the `ROOT` task) and should call `exit()` in function `StopExecution()`. If there are any special actions to take during a reset, overload `Reset()`. If things need to be done (objects deleted or such) before exiting the task, overload `DeleteAllSimulationObjects()`. The `Controller` should also initialize all network interfaces (call `ci_init()` of the network driver) that are in use. The application will assume that all networks from which it can receive messages are initialized.

By the way: in hardware-based simulation, you have to start the program on every node. So you have several instances of the simulation, which are all controlled by `EvalSys`. Since each hardware controller is only responsible for its own node, the nodeID parameter should be the same for all applications created on that node. I recommend storing the nodeID that `EvalSys` sends with the first request for a new application, and checking subsequent commands with a nodeID against the first ID stored. Maybe send an error report if the IDs differ. Please see `cont_hw.cpp` for my own implementation, which will have to be appended in several cases.

CHardwareGps: [gps_hw.cpp/hpp]
The hardware version of the GPS access class, derived from `ACGps` in file `gpsbase.hpp`. Here you have to implement functions `GetGpsTime()`, `GetGpsNumber()`, `EnableGps()`, `DisableGps()`, and `ClearGpsInterrupt()`. The GPS time information is available through an interface, usually RS-232. A template is used to obtain an instance of the GPS class, so do not forget to implement the function `CGpsTemplate::NewGPS()`.

F.7 Tasks And Interrupts

Root Task (Controller): In the `ROOT` task, you should create and install the error report handler and then create a hardware controller and call `Body()`. Note: do not forget to install the error handler, otherwise no error output is possible (even though an `Assert()` still aborts program execution).

code example:

```
// the main task should look like this

#include "cont_hw.hpp"
#include "error.hpp"
#include "err2eval.hpp"  // error handler for output to the evaluation
#include "sockinet.hpp"

#define StreamHost ...  // internet host address of the stream socket
```

```
#define StreamPort ...  // port address (stream socket)

#define DatagramHost ...  // internet host address of the datagram socket
#define DatagramPort ...  // port address (datagram socket)

CErrorEval *errorHandler = NULL;  // error handler
// must be global for the error report task


int main ()
{
    CHardwareController *controller;  // simulation controller
    CInternetAddress addrStream (StreamHost, StreamPort);
    // internet stream socket to the evaluation
    CInternetAddress addrDatagram (DatagramHost, DatagramPort);
    // internet datagram socket for error messages to the evaluation

    // create and install the error handler
    q_create ("QERR", 0, 0, &qid);  // create the error queue
    errorHandler = new CErrorEval (qid, &addrDatagram);
    // create the error handler
    Assert (errorHandler != NULL);
    InstallErrorHandler (errorHandler);  // install it

    // start the error report task (see next item)
    t_create (...);
    t_start (...);

    // create and execute the controller
    q_create ("CTRL", 0, 0, &qid);  // create the controller queue
    controller = new CHardwareController (qid, &addrStream);
    Assert (controller != NULL);
    controller->Body();  // will never return

    exit (0);
}
```

Error Report Task (ErrorHandler): All error outputs of the simulation are done using the three macros `Assert()`, `Check()` and `Report()`. These macros assemble an error string and pass it to an error handler (if it was installed by calling function `InstallErrorHandler()` [`error.hpp`]). Since the string cannot be printed to the screen or passed on to the evaluation at once (because the macros may be called in the context of ISRs), the

handler used for hardware-based simulation stores the message in a queue for later processing. So you should create and install the error handler in `main()` and then start a task which calls `errorHandler->Body()`, causing the error handler to periodically report the error messages in its queue.

The code below uses the handler `CErrorEval` [`err2eval.hpp`], which sends all error messages to the evaluation (over a datagram socket). If a terminal for local output is available and if you prefer local output, use class `CErrorTask` [`errtask.hpp`] instead.

code example:

```
// the error report task should look like this

#include "error.hpp"
#include "err2eval.hpp"

extern CErrorEval *errorHandler;  // error handler

void ErrorTask ()
{
    Assert (errorHandler != NULL);
    errorHandler->Body();  // will never return

    exit (0);
}
```

Application Tasks (Supervisor): Each `Supervisor` is a task, so there should be a global function that can be used for a task. The function needs to know the nodeID/area to find its `Supervisor` in the (global) `listOfSupervisors` and should call `Body()`.

code example:

```
// the application tasks should look like this

#include "error.hpp"
#include "globals.hpp"
#include "supervis.hpp"

void ApplTask (INT32 nodeID, INT32 area)
{
    CSupervisor *sup = listOfSupervisors.Find (nodeID, area);
    Assert (sup != NULL);
```

```
    sup->Body();  // will never return

    exit (0);
}
```

interrupt handling: Interrupts may originate from the LANCE, from the UTCSU, or from the GPS receiver(s). Each interrupt should invoke a certain ISR. Whereas the interrupt of the LANCE is serviced by its driver (so you do not need to install an ISR for it, just provide the `ci_announce()` function), the UTCSU and GPS interrupts require the installation of ISRs.

Unfortunately, the ISR functions responsible for handling these interrupts are not global, but are member functions of objects. Therefore, you cannot simply call the pSOS^{+m} function `set_intvec()` to set interrupt vectors to the ISRs. Rather, you have to install global wrapper functions and call the object ISRs from these global ISRs[15].

File `interupt.hpp` provides a means to call the ISR of an object, if the interrupt vector is known. So you should install global ISRs for all possible interrupt vectors, and these should call the appropriate object ISRs using `InvokeISR()`.

In the case of the UTCSU, which has three different types of interrupts that can occur at the same time and are coded into the interrupt vector, you will have to decide which object ISR to call if interrupts of more than one type are pending. I recommend the following priority for UTCSU interrupt types: INTT (duty timer B of SSU4) before INTN (GPS1PPS interrupt, currently not used) before INTA (hardware snapshot).

By the way: The object ISRs always assert that the interrupt they expect is the only one pending (of the type that the ISR services). If this is a problem, remove the `Assert (!...InterruptPending(INT`xx`))` statement in functions `CClockSync::isr()` [`clocksyn.cpp`] and `CClockInterface::isr()` [`clockifc.cpp`].

Note: for safety purposes, an addition to `SetISR()` (this function is like a `set_intvec()` for object ISRs), which checks the vector and ensures that a global ISR exists, may be reasonable.

code example:

```
// In main(), initialization:
```

[15]Installation of the object ISRs is done by the simulation. You can assume that every interrupt that is raised has an object ISR associated with it.

```
#define VectorGPU1_0 ...
// GPS interrupt vector for the receiver connected to GPU1 of
// the UTCSU on area 0; need defines for all different GPS interrupt
// vectors

#define VectorANT ...  // NTI vector if A, N, and T ints are pending
// need defines for all possible NTI vectors

int main()
{
    void isr_gpu1_0();
    void isr_mod0_intANT();
    // need ISRs for all possible vectors

    set_intvec (VectorGPU1_0, isr_gpu1_0);
    // do that for all GPS interrupt vectors

    set_intvec (VectorANT, isr_mod0_intANT);
    // do that for all NTI modules, all combinations of interrupt vectors
}

// ISRs themselves
void isr_gpu1_0()
{
    InvokeISR (VectorGPU1_0):
    // just call the object ISR responsible for that vector
}

void isr_mod0_intANT()
{
    InvokeISR (VectorT);  // if T has the highest priority
    // call the appropriate object ISR
}
```

References

[1] SynUTC-Homepage:
http://www.auto.tuwien.ac.at/~kmschoss/synutc.html

[2] Günther Gridling: *Evaluation System for a Clock Synchronization Algorithm*
Vienna University of Technology, Dept. of Automation, 1996/97
See the SynUTC-Homepage (Simulation Experiments)

[3] Martin Horauer, Ulrich Schmid, Klaus Schossmaier: *NTI: A Network Time Interface M-Module for High-Accuracy Clock Synchronization*
See the SynUTC-Homepage (Prototype)

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

[5] 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

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

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

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

[9] 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

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

Further 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 should find a floppy disk attached to the rear cover of this thesis. This disk should contain two files: an ASCII version of this installation text (`readme.txt`), and the archive itself (file `simutc.tgz`). Follow these steps for installing the project:

mcopy a:simutc.tgz simutc.tar.gz copies the file from the floppy disk to your current working directory.

tar -xz -f simutc.tar.gz uncompresses the archive and installs the project files. The command will create a directory `SimUTC` (and several sub-directories) 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.