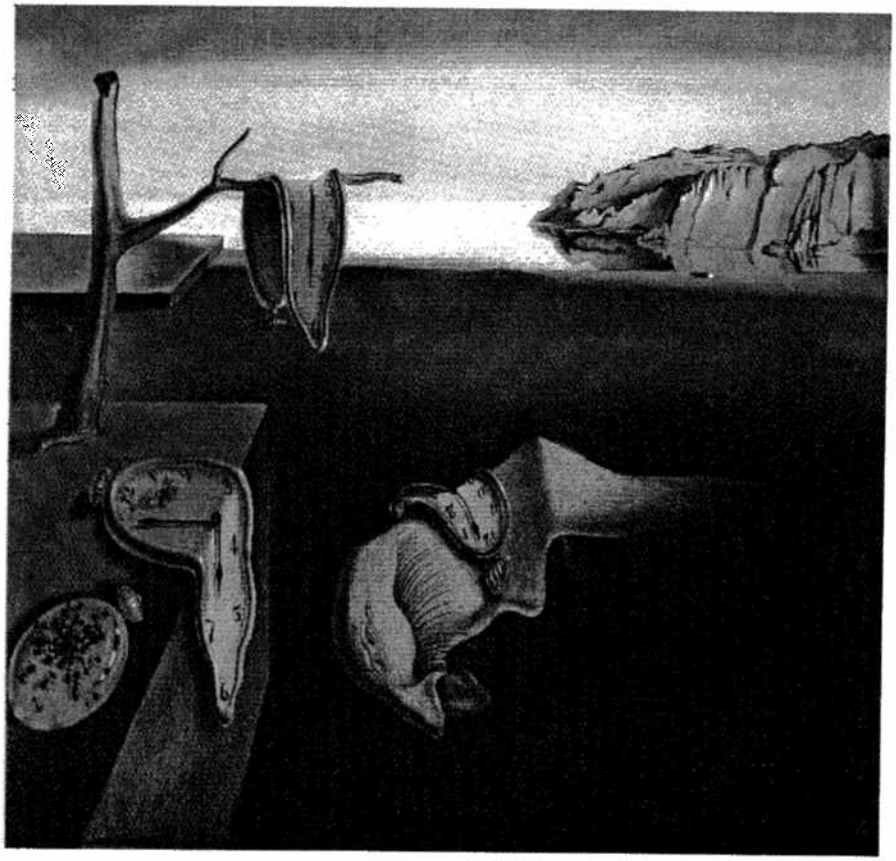
Institut für Automation
Abt. für Automatisierungssysteme

Technische
Universität
Wien

Projektbericht Nr. 183/1-91
February 1999

Implementing the Optimal Precision Algorithm for Clock State&Rate Synchronization

Klaus Schossmaier and Johann Klasek



Salvador Dali, "Die Beständigkeit der Erinnerung"

Implementing the Optimal Precision Algorithm for Clock State&Rate Synchronization

KLAUS SCHOSSMAIER and JOHANN KLASEK

Technische Universität Wien
Department of Automation E183/1
Treitlstraße 1, A-1040 Vienna, Austria
{kmschoss, jk}@auto.tuwien.ac.at

February 16, 1999

Abstract

This report describes how we implemented our Optimal Precision (OP) Algorithm designed for the synchronization of both the state and rate of clocks in a distributed system. The remarkable features of this algorithm are as follows: well-defined intervals are used as means to achieve the synchronization, parameters are required for applying the convergence function, and a state/rate adjustable interval-clock is accessed. The resulting C++ code is integrated in our simulation toolkit SimUTC, where many classes, data structures and functions are available for reuse. In order to test the algorithm, several files are written during a simulation run, which are further processed by the gnuplot tool.

Keywords: clock synchronization, convergence functions, intervals, C++, gnuplot.

1 Introduction

Within the scope of project SynUTC[†] we have developed a simulation toolkit SimUTC (see [Wei97] and [Wei99]) to study the performance of round-based clock sychronization algorithms in a networked distributed system. The desired system behavior can be achieved by setting many parameters concerning the network and the clocks. Here we focus on SimUTC's feature to provide an extension for additional algorithms written in C++ atop of a pre-defined interface to the clock and network.

The algorithm to be implemented has its origins in [SS97], where we devised a generic interval-based *clock state synchronization algorithm* ensuring the accuracy and precision requirement. For a synchronization in the 1 μs-range we found out that the clock rates need to be addressed as well resulting in the framework of [Sch97b]. Similarly, that paper develops a generic interval-based *clock rate synchronization algorithm* ensuring the drift and consonance requirement. Both algorithms lack a particular convergence function, hence we plugged in the *Optimal Precision* (OP) convergence function introduced in [Sch97a]. The final working algorithm is built and analyzed in [Sch98] by combining the clock state and rate synchronization algorithm. In the sequel we call it *OP algorithm.*

[†]Project SynUTC is supported by the Austrian Science Foundation (FWF) under contract no. P10244-ÖMA and is now continued under the START programme Y41-MAT. For further project information take a look at http://www.auto.tuwien.ac.at/Projects/SynUTC/

2 Algorithm

Here we reprint the appropriate portions from the underlying papers [SS97], [Sch97b], [Sch97a], and [Sch98], which are relevant for the implementation of our OP algorithm.

Definition 1 (Initial Synchronization Conditions) *At each node q, the local interval clock $\boldsymbol{C}_q$ has to be initialized to the accuracy interval $\boldsymbol{A}_q^0 = [T_q^0 - \alpha_q^{0-}, T_q^0 + \alpha_q^{0+}]$ at some synchronous real-time t_q^0 by some external means. This state-initialization must ensure*

- $t_q^0 \in \boldsymbol{A}_q^0$,
- $T_q^0 \in [\Lambda + \Omega + \Delta + E_q \pm \pi]$,
- $\boldsymbol{\alpha}_q^0 \subseteq \pi_0$,

where π and π_0 is provided by the analysis. In addition, the rate-initialization has to make sure that each local rate interval $\boldsymbol{R}_q$ maintained at node q is correct and γ^0-correct with $\gamma^0 \subseteq \gamma^*$ at t_q^0, where γ^* is provided by the analysis. Note that π, π_0, and γ^* depend on the particular convergence function employed.

Definition 2 (Generic State&Rate Algorithm) *Periodically, each node q in the system performs the following operations:*

(S) Send: *At $k_S P_S$ calculate the reduced state round $\bar{k}_S = k_S \bmod m$. If $\bar{k}_S > 0$ then initiate broadcast of timestamped packets $\langle T_q, \boldsymbol{\alpha}_q \rangle$ else of $\langle T_q, \boldsymbol{\alpha}_q, \boldsymbol{R}_q, U_q \rangle$.*

(R) Reception and Preprocessing: *Until $k_S P_S + \Lambda + \Omega + \Delta$ timestamp the arriving packets $\langle T_p, \boldsymbol{\alpha}_p \rangle$ or $\langle T_p, \boldsymbol{\alpha}_p, \boldsymbol{R}_p, U_p \rangle$ from nodes p with T_q^p. Then compute the accuracy interval*

$$\boldsymbol{A}_q^p = \begin{cases} \boldsymbol{A}_p + [\delta_{qp} \pm (2\boldsymbol{G}_A + \boldsymbol{\varepsilon}_{qp})] \\ \quad + (k_S P_S + \Lambda + \Omega + \Delta + E_q - T_q^p)\boldsymbol{R}_q + \boldsymbol{u}_q + \overline{\boldsymbol{G}} & \text{if } p \neq q \\ \boldsymbol{A}_q + (k_S P_S + \Lambda + \Omega + \Delta + E_q - T_q^q)\boldsymbol{R}_q & \text{otherwise} \end{cases}$$

and store it in an ordered set $\boldsymbol{\mathcal{A}}_q$. If $\bar{k}_S = 0$ then compute the quotient rate interval

$$\mathbf{Q}_{q,p} = \left[\frac{T_p - T_p^{\prec} + U_p}{T_q^p - T_q^{p,\prec} + U_q} \pm \left(\frac{(\sigma_q + \sigma_p)(T_p - T_p^{\prec} + U_p)}{2} + \frac{\varepsilon_{qp}}{T_q^p - T_q^{p,\prec} + U_q} \right) \right]$$

for $q \neq p$, preserving $T_p^{\prec} = T_p$ and $T_q^{p,\prec} = T_q^p$, the remote rate interval

$$\boldsymbol{R}_{q,p} = \begin{cases} \mathbf{Q}_{q,p} \cdot \mathbf{R}_p + [0 \pm (\sigma_p + \sigma_q)(\Lambda + \Omega + \Delta + E_q)] & \text{if } p \neq q \\ \mathbf{R}_q & \text{otherwise} \end{cases}$$

and store it in an ordered set $\boldsymbol{\mathcal{R}}_q$.

(C) Computation: *At $k_S P_S + \Lambda + \Omega + \Delta$ apply the convergence function $\boldsymbol{\mathcal{CV}}_{\mathcal{F}_S}(\cdot)$ on $\boldsymbol{\mathcal{A}}_q$ resulting in the new accuracy interval $\boldsymbol{A}_q$. If $\bar{k}_S = 0$ then apply the convergence function $\boldsymbol{\mathcal{CV}}_{\mathcal{F}_R}(\cdot)$ on $\boldsymbol{\mathcal{R}}_q$ resulting in the new rate interval $\boldsymbol{R}_q'$. The new local rate interval $\boldsymbol{R}_q = \mathrm{norm}(\boldsymbol{R}_q') + [0 \pm (\bar{k}_S + 1)\sigma_q P_S]$. Reset $\boldsymbol{\mathcal{A}}_q$ and $\boldsymbol{\mathcal{R}}_q$ to $\emptyset$.*

(T) Termination and Resynchronization: At $k_S P_S + \Lambda + \Omega + \Delta + E_q$ set the local interval clock of node q to $\boldsymbol{A}_q$ and deteriorate it with $\boldsymbol{R}_q$. Let the state adjustment $\Upsilon_q = \mathrm{ref}(\boldsymbol{A}_q) - (k_S P_S + \Lambda + \Omega + \Delta + E_q)$. If $\bar{k}_S = 0$ then adjust the clock rate by setting the coupling factor $S_q = S_q \mathrm{ref}(\boldsymbol{R}'_p)$ and $U_q = \Upsilon_q$ else $U_q = U_q + \Upsilon_q$. Finally, increment state round k_S.

Definition 3 (Convergence Function $\mathcal{OP}$) *Let $\mathcal{I}$ be a set of n compatible accuracy intervals and $\mathcal{J} \subseteq \mathcal{I}$ with $|\mathcal{J}| = n' \leq n$ be the set of non-empty ones among them. Moreover, with $\boldsymbol{I}_o^o \in \mathcal{J}$ denoting the interval originating in the own node's clock, let $\hat{\mathcal{J}} = \{\hat{\boldsymbol{J}}^1, \ldots, \hat{\boldsymbol{J}}^{n'-1}, \check{\boldsymbol{I}}_o^o\}$ be the set of associated precision intervals utilizing a given precision $\boldsymbol{\pi}^H$ for $\hat{\boldsymbol{J}}^i$ and $\boldsymbol{\pi}^o \subset \boldsymbol{\pi}^H$ for $\check{\boldsymbol{I}}_o^o$, where π^{H-}, π^{H+}, π^{o-}, π^{o+} are integer multiples of G_S. For some* a priori *given fault-tolerance parameter f and a maximum correction bound $\Upsilon_{\max}$, the* optimal precision convergence function *$\mathcal{OP}_{n-f}^{\boldsymbol{\pi}^o, \boldsymbol{\pi}^H, \Upsilon_{\max}}(\cdot)$, herein abbreviated by $\mathcal{OP}(\cdot)$, is defined by*

$$\mathrm{ref}(\mathcal{OP}(\mathcal{J})) = \begin{cases} r_n & \text{if } |r_n - r_o| \leq \Upsilon_{\max}, \\ r_o + \Upsilon_{\max} & \text{if } r_n - r_o > \Upsilon_{\max}, \\ r_o - \Upsilon_{\max} & \text{if } r_n - r_o < -\Upsilon_{\max}, \end{cases}$$

where

$$\begin{aligned} r_o &= \mathrm{ref}(\boldsymbol{I}_o^o), \\ r_n &= \boldsymbol{\pi}^o\text{-center}_{G_S}\left(\mathcal{M}_{n'}^{n-f}(\hat{\mathcal{J}}) \cap \check{\boldsymbol{I}}_o^o\right), \end{aligned}$$

and

$$\mathcal{OP}(\mathcal{J}) = \left(\mathcal{M}_{n'}^{n-f}(\mathcal{J}) \cap \boldsymbol{I}_o^o\right) \cup \left[\mathrm{ref}(\mathcal{OP}(\mathcal{J})) \pm \mathbf{0}\right]$$

Definition 4 (Marzullo Function) *Given a set $\mathcal{I} = \{\boldsymbol{I}_1, \ldots, \boldsymbol{I}_n\}$ of $n \geq 1$ non-empty compatible intervals with at least $n - f \geq 1$ of the intervals being accurate, $\mathcal{M}_n^{n-f}(\mathcal{I})$ is defined as the largest interval whose edges lie in the intersection of at least $n - f$ different $\boldsymbol{I}_j$'s.*

Definition 5 (Parameter Settings) *When $\mathcal{CV}_{\mathcal{F}_S}(\cdot) \equiv \mathcal{OP}_{n-f}^{\boldsymbol{\pi}^o, \boldsymbol{\pi}^H, \Upsilon_{\max}}(\cdot)$ and $\mathcal{CV}_{\mathcal{F}_R}(\cdot) \equiv \mathcal{OP}_{n-f}^{\Upsilon^o, \Upsilon_H, \Theta_{\max}}(\cdot)$ the analysis provides the following conditions on the parameters:*

- *Computation delay compensation E_p with*

$$E_p \geq \frac{\eta_p + u_p^-}{1 - \rho_p^-}$$

- *Broadcast delay compensation $\Lambda + \Omega$ with*

$$\Lambda + \Omega \geq \frac{\lambda_{\max} + \omega_{\max} + u_{\max}^-}{1 - \rho_{\max}^-}$$

- *Transmission delay compensation Δ with*

$$\begin{aligned} \Delta \geq\ & 2\varepsilon_{\max} + \varepsilon_{\max}^+ + (H+3)u_{\max} + 2G + G_S + \delta_{\max}(1 + \rho_{\max}^-) \\ & + (2P_S + \Lambda + \Omega + 2E_{\max} - 2E_{\min} - 2\delta_{\min})\rho_{\max} \end{aligned}$$

- *State resynchronization period P_S with*

$$P_S \geq \Lambda + \Omega + \Delta + E_{\max}$$

- *Maximum state correction $\Upsilon_{\max}$ with*

$$\Upsilon_{\max} \geq P_S \rho_{\max} + u_{\max}$$

- *Initial precision interval $\boldsymbol{\pi}_0 = [-\pi_0^-, \pi_0^+]$ with*

$$\begin{aligned}
\pi_0^- &= \frac{1}{2}\Big(2\varepsilon_{\max} + (H+2)u_{\max} + 2G + G_S \\
&\quad + (P + \Lambda + \Omega + \Delta + 2E_{\max} - E_{\min} - 2\delta_{\min})\rho_{\max} \\
&\quad + P(\rho_{\max}^+ - \rho_{\max}^-) + u_{\max}^+ - u_{\max}^-\Big) \\
\pi_0^+ &= \frac{1}{2}\Big(2\varepsilon_{\max} + (B+2)u_{\max} + 2G + G_S \\
&\quad + (P + \Lambda + \Omega + \Delta + 2E_{\max} - E_{\min} - 2\delta_{\min})\rho_{\max} \\
&\quad - P(\rho_{\max}^+ - \rho_{\max}^-) - u_{\max}^+ + u_{\max}^-\Big)
\end{aligned}$$

- *Maximum precision interval $\boldsymbol{\pi}^H = [-\pi^{H-}, \pi^{H+}]$ with*

$$\begin{aligned}
\pi^{H-} &= \pi_0^- + 2u_{\max}^- + G + \varepsilon_{\max}^- + (P + E_{\max} - E_{\min} - \delta_{\min})\rho_{\max}^- \\
\pi^{H+} &= \pi_0^+ + 2u_{\max}^+ + \varepsilon_{\max}^+ + (P + E_{\max} - E_{\min} - \delta_{\min})\rho_{\max}^+
\end{aligned}$$

- *Own precision interval $\boldsymbol{\pi}^o = [-\pi^{o-}, \pi^{o+}]$ with*

$$\begin{aligned}
\pi^{o-} &= \pi_0^- + u_{\max}^- + P\rho_{\max}^- \\
\pi^{o+} &= \pi_0^+ + u_{\max}^+ + P\rho_{\max}^+
\end{aligned}$$

- *Rate resynchronization period P_R with*

$$P_R = \left\lceil \sqrt{\frac{2\epsilon_{\max}}{3\sigma_{\max}}} + B + \epsilon_{\max} + 4(F+E)\rho_{\max} \right\rceil_{P_S}$$

- *Maximum rate correction $\Theta_{\max}$ with*

$$\Theta_{\max} \geq 2\sigma_{\max} P_R$$

- *Initial consonance interval $\boldsymbol{\gamma}^*$ with*

$$\boldsymbol{\gamma}^* = \left[0 \pm \left(2\sigma_{\max}\left(P_R + 2(F+E) + \frac{3}{2}B\right) + \frac{2\epsilon_{\max}}{P_R - B - \epsilon_{\max} - 4(F+E)\rho_{\max}}\right)\right]$$

- *Maximum consonance interval $\boldsymbol{\gamma}_H$ with*

$$\boldsymbol{\gamma}_H = \left[0 \pm \left(\sigma_{\max}\left(3P_R + 6(F+E) + 4B\right) + \frac{3\epsilon_{\max}}{P_R - B - \epsilon_{\max} - 4(F+E)\rho_{\max}}\right)\right]$$

- *Own consonance interval $\boldsymbol{\gamma}^o$ with*

$$\boldsymbol{\gamma}^o = \left[0 \pm \left(\sigma_{\max}\left(3P_R + 4(F+E) + 3B\right) + \frac{2\epsilon_{\max}}{P_R - B - \epsilon_{\max} - 4(F+E)\rho_{\max}}\right)\right]$$

3 Programming

This section documents how we transformed the algorithm from Section 2 into C++ code suitable for the SimUTC toolkit. We assume a running version of the latter, see [Wei99] for installation.

3.1 Orientation

Figure 1 provides an overview which files of the SimUTC toolkit need to be written or modified. For hooking up a synchronization algorithm, some additions have to be made in files **general.hpp**, **globals.cpp**, and **Imakefile**, see Section 3.2 for details. The two major files for the implementation of our OP algorithm are **algrate.cpp** and **algrate.hpp** complemented by files **alggen.cpp** and **alggen.hpp**, see Sections 3.3–3.10 for an in-depth explanation. All the extensions made on the so-called "dummy" evaluation system, located in subdirectory `Evalsys`, are to produce data files in subdirectory `gnuplot` during a simulation run, see Section 4 for details and a sample run.

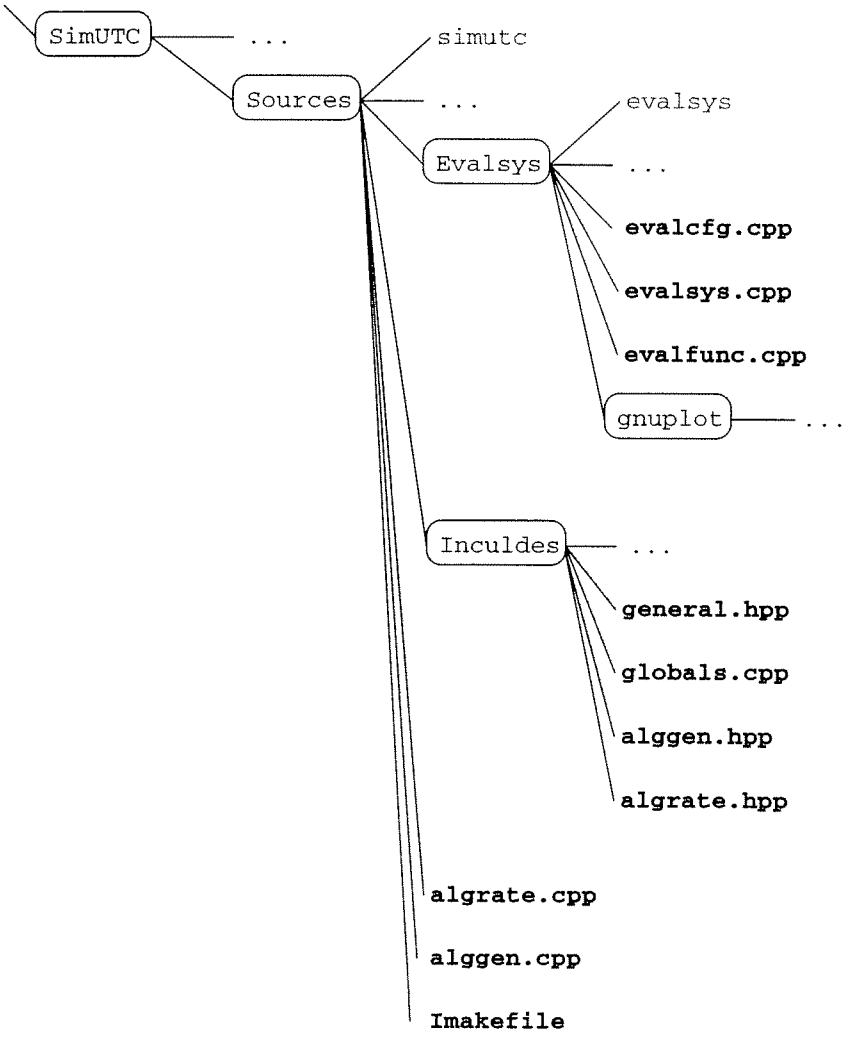


Figure 1: Relevant SimUTC Files

3.2 Hooking up the Algorithm

This is the easiest part, since we just need to inform the system that another algorithm is present. More specifically, in file **general.hpp** the enumeration `EAlgorithmType` is supplied with `Rate`:

```
// possible algorithm types
enum EAlgorithmType
{
    FreeRun = 0,  // no synchronization at all, lets the clock drift
    FaultTolerantAverage,  // fault tolerant average (takes reference point)
    Midpoint,  // like fault tolerant average (but takes midpoint)
    Fetzer,  // like FTA, but P-Nodes are handled separatly
    OP,  // Optimal Precision
    Marzullo,  // Marzullo-Algorithm
    Rate,  // Optimal Precision with rate synchronization
    FaultTolerantIntersection,  // FTI algorithm

    EAlgorithmTypeCount
};
// the last entry EDistributionCount is used for accounting purposes
// and has to stay last even if new items are added!
```

Since our algorithm is implemented as class `CRate` defined in file **algrate.hpp**, the file **globals.cpp** needs to be added with the line

```
#include "algrate.hpp"
```

as well as with the call `algorithmManager.SetFunction (Rate, CRate::CreateRate)` in function `InstallAlgorithms()`:

```
// installs all clock synchronization algorithm types known in the system
// note: if you wish to add a new algorithm type, you must insert its creation
// function into the algorithm manager; just copy the two lines that are
// commented out below, and replace 'YourAlgorithmType' with the enumeration
// constant of your algorithm (you must add one in enum EAlgorithmType in
// file general.hpp), 'YourClassName' with the name of your algorithm
// class (you must have derived one from class ACClockSyncAlgorithm in file
// algbase.hpp) and 'CreateObject' with the name of your create function
// (you must supply your class with such a static function)
void InstallAlgorithms()
{
    algorithmManager.SetFunction (FreeRun, CFreeRun::CreateFreeRun);
    algorithmManager.SetFunction (FaultTolerantAverage, CFta::CreateFta);
    algorithmManager.SetFunction (Midpoint, CMidpoint::CreateMidpoint);
    algorithmManager.SetFunction (Fetzer, CFtaWithFetzer::CreateFtaWithFetzer);
    algorithmManager.SetFunction (OP, COP::CreateOP);
    algorithmManager.SetFunction (Marzullo, CMarzullo::CreateMarzullo);
    algorithmManager.SetFunction (Rate, CRate::CreateRate);
    algorithmManager.SetFunction (FaultTolerantIntersection, CFti::CreateFti);

//    algorithmManager.SetFunction (YourAlgorithmType,
//                                  YourClassName::CreateObject);
}
```

Finally, in file **Imakefile** we need to insert `algrate.o` in `OBJECTS`. Do not forget to start `makedepend` and `makeit` from the shell.

3.3 Defining the Algorithm as Object

Our OP algorithm is implemented as an object of class `CRate` whose definition is contained in file **algrate.hpp**, see Appendix A for the entire listing. The class `CRate` is derived from class `ACClockSyncAlgorithm` along with the following declarations:

- creating function `CreateRate()`
- constructor function `CRate()`
- virtual functions `ConvergenceFunction()`, `Init()`, `ClockChanged()`, `StoreTime()`, `StoreDelay()`, and `StoreRate()`
- member data `cntDelay`, `nodeID`, `area`, `piHMinus`, `piHPlus`, `piOwnMinus`, `piOwnPlus`, `ups`, `sigma`, `gammaHMinus`, `gammaHPlus`, `gammaOwnMinus`, `gammaOwnPlus`, `thetaMax`, and `rateRound`
- member functions `State()` and `Rate()`

The following subsections provide an instruction about the implementation of the above functions, which is done mostly in file **algrate.cpp**, see Appendix B for the entire listing.

3.4 Initializing the Algorithm

Implementing function `CreateRate()` within file **algrate.cpp** is straightforward. Also the constructor function `CRate()` is easy to understand, since it

- initializes the status of the algorithm,
- resets the clock, and
- computes the parameters of our OP algorithm.

The status of the algorithm can be changed by accessing member data `cntDelay` (how many delay measurements took place? initially 0), `rateRound` (is this a rate round? initially not), `nodeID` as well as `area` (what is my node identification? call function `GetOwnID()`), and by calling function `SetResynchronizationStatus()` (is my own time valid? initially not). In case of a SimUTC system reset, function `Init()` is called that re-initializes the status of the algorithm.

The remaining two responsibilities are delegated to function `ClockChanged()`, hence it sets the clock accuracy interval along with its deterioration and computes the necessary parameters. The accuracy interval of the clock is set to the module global constants `accMinusInitial` and `accPlusInitial` via function `SetTimestamp()` together with the conversion function `IntervalToClock()`. The latter is defined resp. implemented within file **alggen.hpp** resp. **alggen.cpp**, see Appendix C resp. D for the entire listing. The deterioration of the accuracy interval is set to the module global constants `thetaPlusInitial` and `thetaMinusInitial` via function `SetLambdapure()` together with the conversion function `ThetaToLambda()` also contained in file **algrate.cpp**. Since the algorithm itself needs to know the deterioration values forming the local rate interval $\boldsymbol{R}_q$, we have to set the global structure `rate->thetaMinus` and `rate->thetaPlus` accordingly.

The parameters are of special interest for our OP algorithm, since they have a great influence on its correctness and performance. When looking back at Section 2, we need to model the following items:

- Maximum precision interval π^H: member data `piHMinus` and `piHPlus`
- Own precision interval π^o: member data `piOwnMinus` and `piOwnPlus`
- Maximum state correction $\Upsilon_{\max}$: member data `ups`
- Maximum oscillator stability $\sigma_{\max}$: member data `sigma`
- Maximum consonance interval γ_H: member data `gammaHMinus` and `gammaHPlus`
- Own consonance interval γ^o: member data `gammaOwnMinus` and `gammaOwnPlus`
- Maximum rate correction $\Theta_{\max}$: member data `thetaMax`

Inside function `ClockChanged()` the above parameters can be computed or set directly with the help of the respective macros `PI_H`, `PI_O`, `UPSILON_MAX`, `SIGMA_MAX`, `GAMMA_H`, `GAMMA_O`, and `THETA_MAX`.

To support the computation of the parameters according to Definition 5 (with some minor simplifications), we make the following inquiries: delay compensations $\Lambda + \Omega + \Delta$ via function `GetDelayCompensation()`, computation delay compensation E_q via function `GetComputationDelayCompensation()`, granularity G via function `GetGranularity()`, state resynchronization period P_S via function `GetRoundPeriod()`, rate resynchronization period P_R via function `GetRatePeriod()`, and maximum clock drifts $\rho^{\pm}_{\max}$ via functions `GetRhoPlus()` and `GetRhoMinus()`. Note that the values for these inquiries come from the evaluation system, see Section 4. For the maximum transmission delay uncertainties $\epsilon^{\pm}_{\max}$ we use the module global constants `epsilonPlusInitial` and `epsilonMinusInitial`, since no delay measurements took place at initialization time. All parameters are printed at run-time when the macro `RUN_SETTING` is defined.

3.5 Programming the Algorithm: (S) Send

The "Send" item of Definition 2 is already built in by the SimUTC system. A duty timer initiates the transmission of a packet that contains the current timestamp T_q as well as accuracy interval $\boldsymbol{\alpha}_q$, and optional the local rate interval $\boldsymbol{R}_q$ given by the global structure `rate->thetaMinus` and `rate->thetaPlus` as well as the sum of applied state adjustments U_q give by the global structure `rate->u`.

3.6 Programming the Algorithm: (R) Reception and Preprocessing

The "Reception and Preprocessing" item of Definition 2 has two parts: one for the state synchronization and the other one for rate synchronization, see Figure 2 for illustration.

The state synchronization part is already built in by the SimUTC system. When such a packet arrives, function `StoreTime()` within file **algbase.cpp** is called, which is responsible for all the preprocessing steps resulting in interval $\boldsymbol{A}^p_q$. These intervals are then stored in a certain database object whose member functions to access it are defined in file **database.hpp**.

The rate synchronization part is implemented in file **algrate.cpp** by overloading function `StoreRate()` whose signature can be found in file **algbase.hpp**. According to item (R) of Definition 2, we first need to compute the quotient rate interval $\boldsymbol{Q}_{p,q}$, represented by `quotientInt` of type `SAsymInterval`, and then the remote rate interval $\boldsymbol{R}_{p,q}$, represented

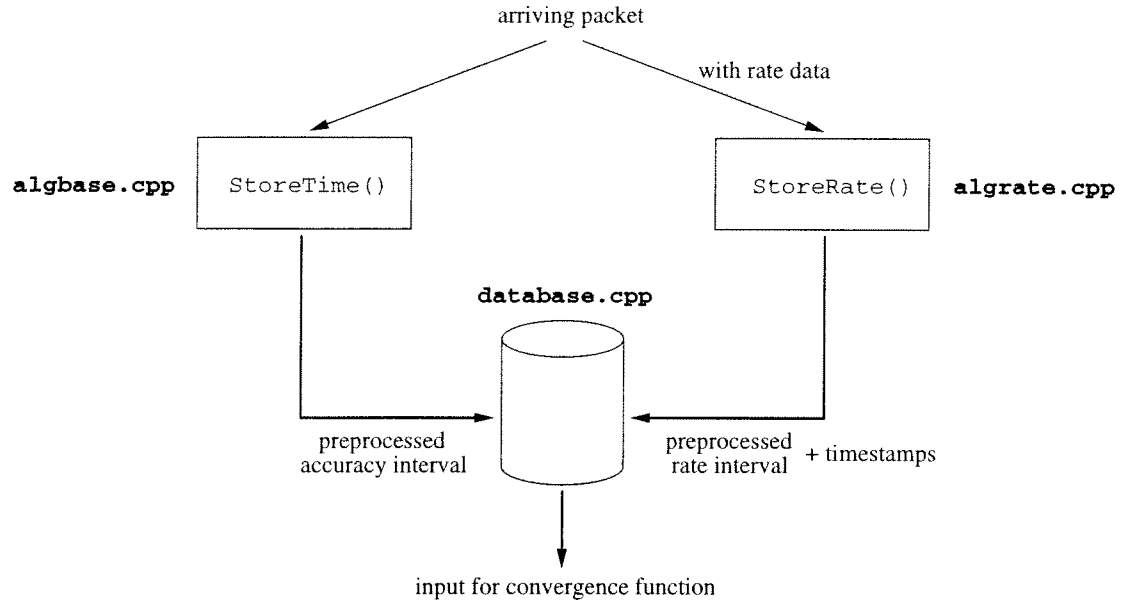


Figure 2: Preprocessing of Accuracy and Rate Intervals

by `rateInt` of the same type. To understand the implementation, we point out that an entry in the designated database is a pointer to an item of type `SRateTime` defined in file **algrate.hpp**. Such an item is capable to hold besides a local rate interval and a sum of applied state adjustments, a reference point and two timestamps. At the beginning, the database is actually filled via function `database->SetRate()`, but later on rate information is read out via function `database->GetRate()` and the corresponding entry is solely modified. Hence the database is exploited in two ways: remembering timestamps to carry out the relative rate measurement and providing the remote rate interval to our convergence function OP.

The arrival of a packet containing rate information is also used to inform the algorithm that a new rate round is emerging. For the sake of simplicity, we have chosen this kind of rate round indication instead of using the round counter k_S as proposed in the algorithm of Definition 2. Thus we set member data `rateRound` to `True`, but not the very first time since it takes at least two packet arrivals to measure the relative rate.

At the end of function `StoreRate()`, the preprocessed rate interval $\boldsymbol{R}_{p,q}$ is reported to the evaluation system via function `ReportRate()`, see Section 4.

3.7 Programming the Algorithm: (C) Computation

The "Computation" item of Definition 2 has also two parts: one for the state synchronization and the other one for rate synchronization, see Figure 3 for illustration. Since the SimUTC system just calls virtual function `ConvergenceFunction()` to initiate the computation, we need to examine member data `rateRound` if a rate synchronization is necessary in addition to a state synchronization. The respective entry points are represented by member functions `Rate()` and `State()` covered in the following two subsections.

Debugging code to display the various intervals, which are all of type `SAsymInterval`, can be inserted via macros `GNU_ACC`, `GNU_PRE`, `GNU_DRIFT`, and `GNU_CONS`, see Section 3.10 for their usage.

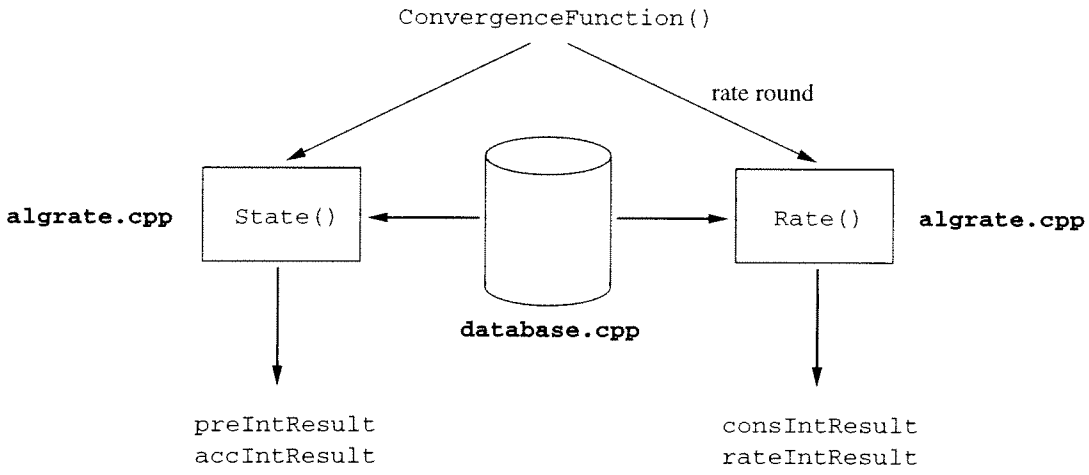


Figure 3: Convergence Function $\mathcal{OP}$ for State and Rate Synchronization

3.7.1 Convergence Function $\mathcal{OP}$ for State Synchronization

Function `State()` within file **algrate.cpp** is in charge of computing the new clock interval given by `point`, `alpha_minus` and `alpha_plus` for the network `networkID` by using the preprocessed intervals $\boldsymbol{A}_q^p = [a_q^p \pm \boldsymbol{\alpha}_q^p]$ that are stored in the database, see Figure 2 and 3. According to Definition 3 of our Convergence Function $\mathcal{OP}$ adapted for clock state synchronization, we have to perform the following steps:

- Associated precision intervals:

$$\begin{aligned} \hat{\boldsymbol{A}}_q^p &:= [a_q^p \pm \boldsymbol{\pi}^H] \text{ for } p \neq q \\ \hat{\boldsymbol{A}}_q^q &:= [a_q^q \pm \boldsymbol{\pi}^o] \text{ otherwise} \end{aligned}$$

  Associated precision intervals $\hat{\boldsymbol{A}}_q^p, p \neq q$ correspond with `precisionIntervals[i]` having an index larger than zero, and the own associated precision interval $\hat{\boldsymbol{A}}_q^q$ with `precisionIntervals[0]`. The precision intervals $\boldsymbol{\pi}^H = [-\pi^{H-}, \pi^{H+}]$ and $\boldsymbol{\pi}^o = [-\pi^{o-}, \pi^{o+}]$ are given by the member data `piHMinus`, `piHPlus`, `piOwnMinus`, and `piOwnPlus`, see Section 3.4.

- Marzullo's function upon associated precision intervals:

$$\hat{\boldsymbol{A}}_q^{\mathcal{M}} := \mathcal{M}_{n'}^{n-f}(\hat{\boldsymbol{A}}_q^1, \ldots, \hat{\boldsymbol{A}}_q^n)$$

  Resulting precision interval $\hat{\boldsymbol{A}}_q^{\mathcal{M}}$ corresponds with `preIntMarz`, number of non-empty intervals n' with `sizeState`, and maximum number of faulty intervals f with `numberOfFaultyNodes`. Marzullo's function $\mathcal{M}$ as shown in Definition 4 is implemented by function `ApplyMarzullo()` contained in file **alggen.cpp**.

- Intersection with own associated precision interval:

$$\hat{\boldsymbol{A}}_q' := \hat{\boldsymbol{A}}_q^{\mathcal{M}} \cap \hat{\boldsymbol{A}}_q^q$$

  Resulting precision interval $\hat{\boldsymbol{A}}_q'$ corresponds with `preIntResult`. The intersection of intervals is implemented by function `Intersect()` contained in file **alggen.cpp**.

- Candidate reference point:

$$a'_q := \frac{\pi^{o+}\text{right}(\hat{\boldsymbol{A}}'_q) + \pi^{o+}\text{left}(\hat{\boldsymbol{A}}'_q)}{\pi^{o-} + \pi^{o+}}$$

Reference point a'_q corresponds with `preIntResult.ref`. It was first the midpoint and is now weighted by member data `piOwnMinus` and `piOwnPlus`, see Section 3.4.

- New reference point:

$$a_q(\mathcal{OP}) := \begin{cases} a'_q & \text{if } |a'_q - a^q_q| \leq \Upsilon_{\max} \\ a^q_q + \Upsilon_{\max} & \text{if } a'_q - a^q_q > \Upsilon_{\max} \\ a^q_q - \Upsilon_{\max} & \text{if } a'_q - a^q_q < -\Upsilon_{\max} \end{cases}$$

Reference point $a_q(\mathcal{OP})$ corresponds again with `preIntResult.ref`, difference $a'_q - a^q_q$ with local variable `adjustment`, and maximum state correction $\Upsilon_{\max}$ with member data `ups`, see Section 3.4. The actual state adjustment gets accumulated by the global structure `rate->u` as specified by item (T) of Definition 2.

- Marzullo's function upon accuracy intervals:

$$\boldsymbol{A}^{\mathcal{M}}_q := \mathcal{M}^{n-f}_{n'}(\boldsymbol{A}^1_q, \ldots, \boldsymbol{A}^n_q)$$

Accuracy intervals $\boldsymbol{A}^p_q$ correspond with `accuracyIntervals[]` and accuracy interval $\boldsymbol{A}^{\mathcal{M}}_q$ with `accIntMarz`. Marzullo's function $\mathcal{M}$ is implemented as said above.

- Intersection with local accuracy interval:

$$\boldsymbol{A}'_q := \boldsymbol{A}^{\mathcal{M}}_q \cap \boldsymbol{A}^q_q$$

Accuracy interval $\boldsymbol{A}'_q$ corresponds with `accIntResult` and local accuracy interval $\boldsymbol{A}^q_q$ with `accuracyIntervals[0]`. The intersection of intervals is implemented as said above.

- New accuracy interval endpoints:

$$\begin{aligned} \text{right}(\boldsymbol{A}_q(\mathcal{OP})) &:= \max\{\text{right}(\boldsymbol{A}'_q), a_q(\mathcal{OP})\} \\ \text{left}(\boldsymbol{A}_q(\mathcal{OP})) &:= \min\{\text{left}(\boldsymbol{A}'_q), a_q(\mathcal{OP})\} \end{aligned}$$

Endpoints $\text{right}(\boldsymbol{A}_q(\mathcal{OP}))$ and $\text{right}(\boldsymbol{A}'_q)$ correspond with `accIntResult.right`, endpoints $\text{left}(\boldsymbol{A}_q(\mathcal{OP}))$ and $\text{left}(\boldsymbol{A}'_q)$ with `accIntResult.left`, and the already computed reference point $a_q(\mathcal{OP})$ with `preIntResult.ref`.

3.7.2 Convergence Function $\mathcal{OP}$ for Rate Synchronization

Function `Rate()` within file **algrate.cpp** is in charge of computing the multiplication factor `multStep` for the coupling factor and the new local rate interval given by `theta_minus` and `theta_plus` for the network `networkID` by using the preprocessed intervals $\boldsymbol{R}_{q,p} = [\vartheta^-_{q,p}, r_{q,p}, \vartheta^+_{q,p}]$ that are stored in the database, see Figure 2 and 3. Moreover, the sum of state adjustment U_q given by global structure `rate->u` needs to be reset. According to Definition 3 of our Convergence Function $\mathcal{OP}$ adapted for clock rate synchronization, we have to perform the following steps:

- Associated consonance intervals:

$$\begin{aligned} \boldsymbol{\gamma}_{q,p} &:= [\gamma_H^-, r_{q,p}, \gamma_H^+] \text{ for } p \neq q \\ \boldsymbol{\gamma}_{q,q} &:= [\gamma^{o-}, r_{q,q}, \gamma^{o+}] \text{ otherwise} \end{aligned}$$

Associated consonance intervals $\boldsymbol{\gamma}_{q,p}, p \neq q$ correspond with `consonanceIntervals[i]` having an index larger than zero, and the own associated consonance interval $\boldsymbol{\gamma}_{q,q}$ with `consonanceIntervals[0]`. As before, the consonance intervals $\boldsymbol{\gamma}_H = [\gamma_H^-, 0, \gamma_H^+]$ and $\boldsymbol{\pi}^o = [\gamma^{o-}, 0, \gamma^{o+}]$ are available as member data `gammaHMinus`, `gammaHPlus`, `gammaOwnMinus`, and `gammaOwnPlus`, see Section 3.4.

- Marzullo's function upon associated consonance intervals:

$$\boldsymbol{\gamma}_q^{\mathcal{M}} := \mathcal{M}_{n'}^{n-f}(\boldsymbol{\gamma}_{q,1}, \ldots, \boldsymbol{\gamma}_{q,n})$$

Resulting consonance interval $\boldsymbol{\gamma}_q^{\mathcal{M}}$ corresponds with `consIntMarz`, number of non-empty intervals n' with `sizeRate`, and maximum number of faulty intervals f with `numberOfFaultyNodes`. Marzullo's function $\mathcal{M}$ as shown in Definition 4 is implemented by function `ApplyMarzullo()` contained in file **alggen.cpp**.

- Intersection with own associated consonance interval:

$$\boldsymbol{\gamma}_q' := \boldsymbol{\gamma}_q^{\mathcal{M}} \cap \boldsymbol{\gamma}_{q,q}$$

Resulting consonance interval $\boldsymbol{\gamma}_q'$ corresponds with `consIntResult`. The intersection of intervals is implemented by function `Intersect()` contained in file **alggen.cpp**.

- Candidate reference point:

$$r_q' := \frac{\gamma^{o+}\text{right}(\boldsymbol{\gamma}_q') + \gamma^{o+}\text{left}(\boldsymbol{\gamma}_q')}{\gamma^{o-} + \gamma^{o+}}$$

Reference point r_q' corresponds with `consIntResult.ref`. It was first the midpoint and is now weighted by member data `gammaOwnMinus` and `gammaOwnPlus`, see Section 3.4.

- New reference point:

$$r_q'(\boldsymbol{\mathcal{OP}}) := \begin{cases} r_q' & \text{if } |r_q' - r_{q,q}| \leq \Theta_{\max} \\ r_{q,q} + \Theta_{\max} & \text{if } r_q' - r_{q,q} > \Theta_{\max} \\ r_{q,q} - \Theta_{\max} & \text{if } r_q' - r_{q,q} < -\Theta_{\max} \end{cases}$$

Reference point $r_q'(\boldsymbol{\mathcal{OP}})$ corresponds again with `consIntResult.ref` and maximum rate correction $\Theta_{\max}$ with member data `thetaMax`, see Section 3.4. As specified by item (T) of Definition 2, the new reference point equals the multiplication factor `multStep` for the coupling factor.

- Marzullo's function upon remote rate intervals:

$$\boldsymbol{R}_q^{\mathcal{M}} := \mathcal{M}_{n'}^{n-f}(\boldsymbol{R}_{q,1}, \ldots, \boldsymbol{R}_{q,n})$$

Remote rate intervals $\boldsymbol{R}_{q,p}$ correspond with `rateIntervals[]` and rate interval $\boldsymbol{R}_q^{\mathcal{M}}$ with `rateIntMarz`. Marzullo's function $\mathcal{M}$ is implemented as said above.

- Intersection with local rate interval:

$$\boldsymbol{R}'_q := \boldsymbol{R}_q^{\mathcal{M}} \cap \boldsymbol{R}_{q,q}$$

Rate interval $\boldsymbol{R}'_q$ corresponds with `rateIntResult` and local rate interval $\boldsymbol{R}_{q,q}$ with `rateIntervals[0]`. The intersection of intervals is implemented as said above.

- New rate interval endpoints:

$$\begin{aligned} \text{right}(\boldsymbol{R}'_q(\mathcal{OP})) &:= \max\{\text{right}(\boldsymbol{R}'_q), r'_q(\mathcal{OP})\} \\ \text{left}(\boldsymbol{R}'_q(\mathcal{OP})) &:= \min\{\text{left}(\boldsymbol{R}'_q), r'_q(\mathcal{OP})\} \end{aligned}$$

Endpoints $\text{right}(\boldsymbol{R}'_q(\mathcal{OP}))$ and $\text{right}(\boldsymbol{R}'_q)$ correspond with `rateIntResult.right`, endpoints $\text{left}(\boldsymbol{R}'_q(\mathcal{OP}))$ and $\text{left}(\boldsymbol{R}'_q)$ with `rateIntResult.left`, and the already computed reference point $r'_q(\mathcal{OP})$ with `consIntResult.ref`.

- Normalizing the rate interval:

$$\boldsymbol{R}_q := \text{norm}(\boldsymbol{R}'_q) = \frac{\boldsymbol{R}'_q}{r'_q(\mathcal{OP})}$$

The inverse rate drifts ϑ_q^- resp. ϑ_q^+ of the new local rate interval $\boldsymbol{R}_q = [\vartheta_q^-, 1, \vartheta_q^+]$ correspond with `theta_minus` resp. `theta_plus`. Again, reference point $r'_q(\mathcal{OP})$ is given by `consIntResult.ref`. The gradual deterioration of $\boldsymbol{R}_q$ is deferred to function `ConvergenceFunction()`.

3.8 Programming the Algorithm: (T) Termination and Resynch.

Returning to virtual function `ConvergenceFunction()`, we have to prepare the rate and state adjustments, which are set forth by the SimUTC system at the scheduled point of time, see item "Termination and Resynchronization" of Definition 2.

In case of a rate round, the `STEP`-register of our clock (aka. coupling factor S_q) is read resp. written via function `GetSteppure()` resp. `SetSteppure()`, and the new local rate interval $\boldsymbol{R}_q$ is assigned to the global structure `rate->thetaMinus` and `rate->thetaPlus`. The counterpart for the rate round indication featured by member data `rateRound` is located here as well, see Section 3.6 for an explanation.

In any case, the new rate interval is deteriorated for the next state round, which is necessary for the proper preprocessing of the accuracy intervals. In addition, function `SetLambdapure()` together with the conversion function `ThetaToLambda()` enforce the new deterioration (represented by the abovementioned rate interval) of the local accuracy interval. Module global function `ThetaToLambda()` is contained in file **algrate.cpp**. Assigning `*newClockValue` with the new clock interval given by `point`, `alpha_minus` and `alpha_plus` together with conversion function `IntervalToClock()` prepares the upcoming state correction. Again, the listing of the latter function can be found in Appendix C and D.

Finally, function `ConvergenceFunction()` reports the new accuray interval $\boldsymbol{A}_q$ and the new rate interval $\boldsymbol{R}'_q$ to the evaluation system via function `ReportInterval()` and `ReportRate()`, respectively, see Section 4.

3.9 Improving the Transmission Delay Measurement

At startup time the algorithm has not done any transmission delay measurements, hence it makes sense to postpone the synchronization until some useful transmission delay data have been gathered. For that purpose we overload function `StoreTime()` and `StoreDelay()` accordingly in our file **algrate.hpp** resp. **algrate.cpp**, see Appendix A resp. B for the entire listing. The idea is to use member data `cntDelay` as a counter for the number of transmission delay measurements. When this counter is below the value of the module global constant `StartSynchronization` we are literally dealing with a free-run, otherwise the synchronization becomes active. The status of the algorithm can be changed via function `SetResynchronizationStatus()` and inquired via function `GetResynchronizationStatus()`.

3.10 Debugging the Code

For debugging the wealth of intervals in our file **algrate.cpp** we introduced macros to enable the generation of code that outputs a sequence of intervals on data files. Module global function `PlotInterval()` is used for writing these files whose contents can be displayed with the `gnuplot`-tool. As a result, in conjunction with a source level debugger like `ddd`, we are able to trace the computation of the following intervals:

- Accuracy intervals by defining macro `GNU_ACC` to write them onto a data file located by macro `NAME_ACC`.
- Precision intervals by defining macro `GNU_PRE` to write them onto a data file located by macro `NAME_PRE`.
- Rate intervals by defining macro `GNU_DRIFT` to write them onto a data file located by macro `NAME_DRIFT`.
- Consonance intervals by defining macro `GNU_CONS` to write them onto a date file located by macro `NAME_CONS`.

For example, the intervals in data file **pre.dat** can be transformed into a postscript file **pre.ps** when executing the command "`gnuplot pre.gnu`" by using the following `gnuplot`-script **pre.gnu**:

```
set terminal postscript eps
set output "pre.ps"
set format y "%.6f"
set noyzeroaxis
set xtics 0,1,10
set title "PRECISION INTERVALS"
plot [-1:11] "pre.dat" with errorbars
exit
```

Figure 4 displays the result, where the intervals at x-position $0, \ldots, 4$ are the associated precision intervals $\hat{\boldsymbol{A}}_q^1, \ldots, \hat{\boldsymbol{A}}_q^5$ fed into the convergence function, the one at x-position 6 is the precision interval $\hat{\boldsymbol{A}}_q^{\mathcal{M}}$ after the application of Marzullo's function, the one at x-position 7 is the precision interval $\hat{\boldsymbol{A}}_q'$ after the intersection with the own associated precision interval, and the one at x-position 8 is the final precision interval with the proper reference point.

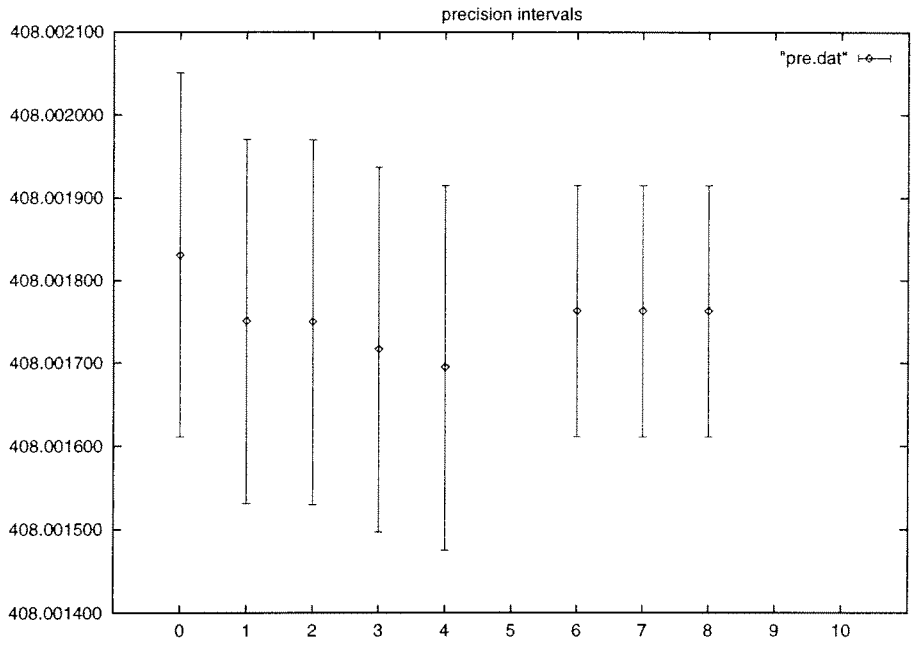


Figure 4: Precision Intervals

4 Evaluation

As already shown in Figure 1, the SimUTC toolkit consists of the simulation program `simutc` in subdirectory `Sources` and the "dummy" evaluation program `evalsys` in subdirectory `Evalsys`. Both programs run autonomously and communicate via TCP/IP stream sockets for system control and event reporting, see Figure 5 for a schematic view. All the simulation parameters (e.g. number of nodes, clock and network characteristics, snapshots) setting up a particular run are provided by file **evalcfg.cpp**, see Appendix F for an example.

The original version of the evaluation system reports some basic results after the simulation terminates. However, to obtain more information of a simulation run, we extended the evaluation system insofar that data files are produced during a run, see Figure 5. More specifically, data file **state*q*.dat** records the traditional accuracy of node q, data file **pik*j*.dat** the j-th precision, data file **adj*q*.dat** the state adjustment of node q, data file **acc*q*.dat** the accuracy interval of node q, data file **rate*q*.dat** the rate deviation of node q, data file **gammak*j*.dat** the j-th consonance, data file **mult*q*.dat** the rate adjustment of node q, and data file **drift*q*.dat** the rate interval of node q. Note that index $q \in \{1, \ldots, N\}$ and $j \in \{0, \ldots, M\}$, where $N = M + 1$ is the number of nodes. Section 4.1 explains how we produce these files and Section 4.2 shows a sample run.

To display the contents of the data files, we use them as input for the `gnuplot`-tool together with the respective scripts **state.gnu**, **pi.gnu**, **adj.gnu**, **acc.gnu**, **rate.gnu**, **gamma.gnu**, **mult.gnu**, and **drift.gnu**. The resulting postscript files are merged in a single file **run.dvi** by running `latex` with wrapper file **run.tex** as input. The latter file is also produced by the evaluation system, thus each run can be attributed with the corresponding simulation parameters. Eventually, `dvips` is called to get the postscript file **run.ps**. All tools can be conveniently invoked by the following shell-script `run`:

```
make_pi_origin
make_gamma_origin
gnuplot state.gnu
gnuplot pi.gnu
gnuplot adj.gnu
gnuplot acc.gnu
gnuplot rate.gnu
gnuplot gamma.gnu
gnuplot mult.gnu
gnuplot drift.gnu
latex run.tex
dvips run.dvi
```

Note that the first two lines `make_pi_origin` and `make_gamma_origin` are simple `perl`-scripts that fix the x-axis to zero for the precision and consonance plots, since the autoscaling of the `gnuplot`-tool cannot be switched off individually for the x-axis. In fact, these scripts produce file **pik0_origin.dat** and **gammak0_origin.dat** out of file **pik0.dat** and **gammak0.dat**, respectively, see Section 4.1 for more details.

Figure 5: Flow of Simulation Data

4.1 Producing the Data Files

This section documents how we extended the evaluation system to output the data files depicted in Figure 5. First of all, to open the necessary file buffers, we just added the following lines in function `main()` located in file **evalsys.cpp**:

```
#ifdef PLOT_DATA
    OpenPlotFiles();
#endif
```

Almost all additions to the evaluation system can be found in file **evalfunc.cpp**, see Appendix E for the entire listing. Here function `PrintCSAEvent()` is of special interest, since it handles the whole range of events, which are generated by the simulation system. The relevant events are as follows:

- Events `ClockUpdate` are handled by using streams `outAdj[q]` for the output of state adjustmens in files **adj*q*.dat**.
- Events `RateEvent` are handled by using streams `outDrift[q]` for the output of rate intervals in files **drift*q*.dat**, and streams `outMult[q]` for the rate adjustments in files **mult*q*.dat**.
- Events `Snapshot` are handled by using streams `outPik[j]` when calling function `AddPrecision()` for the output of precisions in files **pik*j*.dat**, streams `outState[q]` for the traditional accuracies in files **state*q*.dat**, streams `outAcc[q]` for the accuracy intervals in files **acc*q*.dat**, streams `outRate[q]` for the rate deviations in files **rate*q*.dat**, and streams `outGammak[j]` when calling function `AddConsonance()` for the consonances in files **gammak*j*.dat**.

In the following subsections we explain how the above quantities are implemented. In general, function `PlotPoint()` is used to output a pair (time, point) and function `PlotInterval()` to output a pair (time, interval) both in `gnuplot`-format. The simulation time of any reported event is given by `event->time` and `ev->nodeID-1` tells from which node the event originates.

4.1.1 Traditional Accuracy

The traditional accuracy of a clock at node q is defined by $C_q(t) - t$, so its calculation is straightforward given that the current clock value corresponds with `ev->newTime`. Just plotting the state $C_q(t)$ is not useful since differences in the progression among the clocks can only be observed with difficulties. For an example see the left plot in the 1st row of Figure 6. The period of `Snapshot`-events is fixed by `SnapshotPeriod` in file **evalcfg.cpp**. The `gnuplot`-script **state.gnu** for post-processing 5 nodes looks as follows:

```
set terminal postscript eps color solid "Times-Roman"
set output "state.ps"
set format y "%.6f"
set grid
set title "TRADITIONAL ACCURACY [sec]"
plot "state1.dat" with linespoints, "state2.dat" with linespoints,
"state3.dat" with linespoints, "state4.dat" with linespoints,
"state5.dat" with linespoints
exit
```

4.1.2 Precision

The precision of an ensemble of clocks is defined by $\max\{C_q(t) - C_p(t)\}$. It is calculated with the help of function `AddPrecision()`, which saves the traditional accuracy of clocks that belong to the same simulation time. In addition, function `PrecisionKth()` allows to compute the k-th maximum distance between the clock states, which is useful to filter out faulty ones. This leads to the **pik*j*.dat** files, but we usually stick to the case $j = 0$. For an example see the left plot in the 2nd row of Figure 6. Again, the period of `Snapshot`-events is fixed by `SnapshotPeriod` in file **evalcfg.cpp**. The `gnuplot`-script **pi.gnu** for post-processing looks as follows:

```
set terminal postscript eps solid "Times-Roman"
set output "pi.ps"
set format y "%.6f"
set grid
set title "PRECISION [sec]"
plot "pik0_origin.dat" with dots, "pik0.dat" with linespoints
exit
```

The `perl`-script `make_pi_origin` to produce file **pik0_origin.dat** that fixes the x-axis to zero looks like:

```
#!/usr/local/bin/perl

$filename = "pik0.dat";
open(IN,"<$filename") || die "cannot open $filename";
$filename = "pik0_origin.dat";
open(OUT,">$filename") || die "cannot open $filename";
$line = <IN>;
($a1, $a2) = split(/ /,$line);
print OUT "$a1 0.0";

close(IN);
close(OUT);
```

4.1.3 State Adjustment

The state adjustment of a clock at node q is defined by $C_q^{\text{new}}(t) - C_q^{\text{old}}(t)$, where the new clock value corresponds with `ev->newTime` and the old one with `ev->clockTime`. The occurrence of `ClockUpdate`-events is determined by the state round period `RoundPeriod` in file **evalcfg.cpp**. For an example see the left plot in the 3rd row of Figure 6. The `gnuplot`-script **adj.gnu** for post-processing 5 nodes looks as follows:

```
set terminal postscript eps color solid "Times-Roman"
set output "adj.ps"
set format y "%.6f"
set grid
set title "STATE ADJUSTMENTS [sec]"
plot "adj1.dat" with linespoints, "adj2.dat" with linespoints,
"adj3.dat" with linespoints, "adj4.dat" with linespoints,
"adj5.dat" with linespoints
exit
```

4.1.4 Accuracy Interval

The accuracy interval of a clock at node q is defined by $\boldsymbol{A}_q = [a_q \pm \boldsymbol{\alpha}_q]$, where the interval of accuracies $\boldsymbol{\alpha}_q$ is encoded by `ev->newTime.acc`. We (re)use the traditional accuracy as reference point and impose a small fixed x-offset for each node to display all accuracy intervals in a single plot. For an example see the left plot in the 4th row of Figure 6.

Again, the period of Snapshot-events is fixed by SnapshotPeriod in file **evalcfg.cpp**. The gnuplot-script **acc.gnu** for post-processing 5 nodes looks as follows:

```
set terminal postscript eps color solid "Times-Roman"
set output "acc.ps"
set format y "%.6f"
set grid
set title "ACCURACY INTERVALS [sec]"
plot "acc1.dat" with errorbars, "acc2.dat" with errorbars,
"acc3.dat" with errorbars, "acc4.dat" with errorbars,
"acc5.dat" with errorbars
exit
```

4.1.5 Rate Deviation

The rate deviation of a clock at node q is defined by $v_q(t) - 1$. Rate $v_q(t)$ is approximated by $(C_q(t) - C_q(t'))/(t - t')$, where the former clock state $C_q(t')$ is remembered by array `state_old[]` and the former snapshot time t' by array `time_old[]`. Intervening state adjustments are suppressed (i.e. skipped) with the aid of array `inhibit[]`, otherwise spikes would impair the plot. For an example see the right plot in the 1st row of Figure 6. Again, the period of Snapshot-events is fixed by SnapshotPeriod in file **evalcfg.cpp**. The gnuplot-script **rate.gnu** for post-processing 5 nodes looks as follows:

```
set terminal postscript eps color solid "Times-Roman"
set output "rate.ps"
set format y "%.6f"
set grid
set title "RATE DEVIATION [ppm]"
plot "rate1.dat" with linespoints, "rate2.dat" with linespoints,
"rate3.dat" with linespoints, "rate4.dat" with linespoints,
"rate5.dat" with linespoints
exit
```

4.1.6 Consonance

The consonance of an ensemble of clocks is defined by $\max\{v_q(t) - v_p(t)\}$. It is calculated with the help of function `AddConsonance()`, which saves the rate of clocks that belong to the same simulation time. In addition, function `PrecisionKth()` allows to compute the k-th maximum distance between the clock rates, which is useful to filter out faulty ones. This leads to the **gammak*j*.dat** files, but we usually stick to the case $j = 0$. For an example see the right plot in the 2nd row of Figure 6. Again, the period of Snapshot-events is fixed by SnapshotPeriod in file **evalcfg.cpp**. The gnuplot-script **gamma.gnu** for post-processing looks as follows:

```
set terminal postscript eps solid "Times-Roman"
set output "gamma.ps"
set format y "%.6f"
set grid
set title "CONSONANCE [ppm]"
plot "gammak0_origin.dat" with dots, "gammak0.dat" with linespoints
exit
```

The perl-script `make_gamma_origin` to produce file **gammak0_origin.dat** that fixes the x-axis to zero looks like:

```
#!/usr/local/bin/perl

$filename = "gammak0.dat";
```

```
open(IN,"<$filename") || die "cannot open $filename";
$filename = "gammak0_origin.dat";
open(OUT,">$filename") || die "cannot open $filename";
$line = <IN>;
($a1, $a2) = split(/ /,$line);
print OUT "$a1 0.0";

close(IN);
close(OUT);
```

4.1.7 Rate Adjustment

The relative rate adjustment of a clock at node q is defined by $v_q^{\text{new}}(t)/v_q^{\text{old}}(t) - 1$, where the ratio is taken over immediately from the rate algorithm via `rateev->refPoint`. To be more exact, it was the value `multStep` computed by function `Rate()` in file **algrate.cpp**. The occurrence of `RateEvent`-events is determined by the rate round period expressed by the multiplication of `RateCycles` with `RoundPeriod` both in file **evalcfg.cpp**. For an example see the right plot in the 3rd row of Figure 6. The `gnuplot`-script **mult.gnu** for post-processing 5 nodes looks as follows:

```
set terminal postscript eps color solid "Times-Roman"
set output "mult.ps"
set format y "%.6f"
set grid
set title "RATE ADJUSTMENTS [ppm]"
plot "mult1.dat" with linespoints, "mult2.dat" with linespoints,
"mult3.dat" with linespoints, "mult4.dat" with linespoints,
"mult5.dat" with linespoints
exit
```

4.1.8 Rate Interval

The local rate interval of a clock at node q is defined by $\boldsymbol{R}_q = [\vartheta_q^-, 1, \vartheta_q^+]$, where the drift rates correspond with `rateev->rate.thetaMinus` and `rateev->rate.thetaPlus`. We impose a small fixed x-offset for each node as well to display all rate intervals in a single plot like in Section 4.1.4. For an example see the right plot in the 4th row of Figure 6. Again, occurrence of `RateEvent`-events is determined by the rate round period expressed by the multiplication of `RateCycles` with `RoundPeriod` both in file **evalcfg.cpp**. The `gnuplot`-script **drift.gnu** for post-processing 5 nodes looks as follows:

```
set terminal postscript eps color solid "Times-Roman"
set output "drift.ps"
set format y "%.6f"
set grid
set title "RATE INTERVALS [1]"
plot "drift1.dat" with errorbars, "drift2.dat" with errorbars,
"drift3.dat" with errorbars, "drift4.dat" with errorbars,
"drift5.dat" with errorbars
exit
```

4.2 Example

Figure 6 shows the resulting plots of a simulation run controlled by file **evalcfg.cpp** in Appendix F having switched on algorithm `Rate`. Note that the left plots are concerned with clock state synchronization, and the right plots with clock rate synchronization. In a few words, the simulated system has 5 non-faulty nodes, where the clocks have an average drift of 0 ppm, 0.5 ppm, -0.5 ppm, -1 ppm, and 2 ppm. The deterministic network delay is 60 μs and the uncertainty is -10 μs and $+20$ μs.

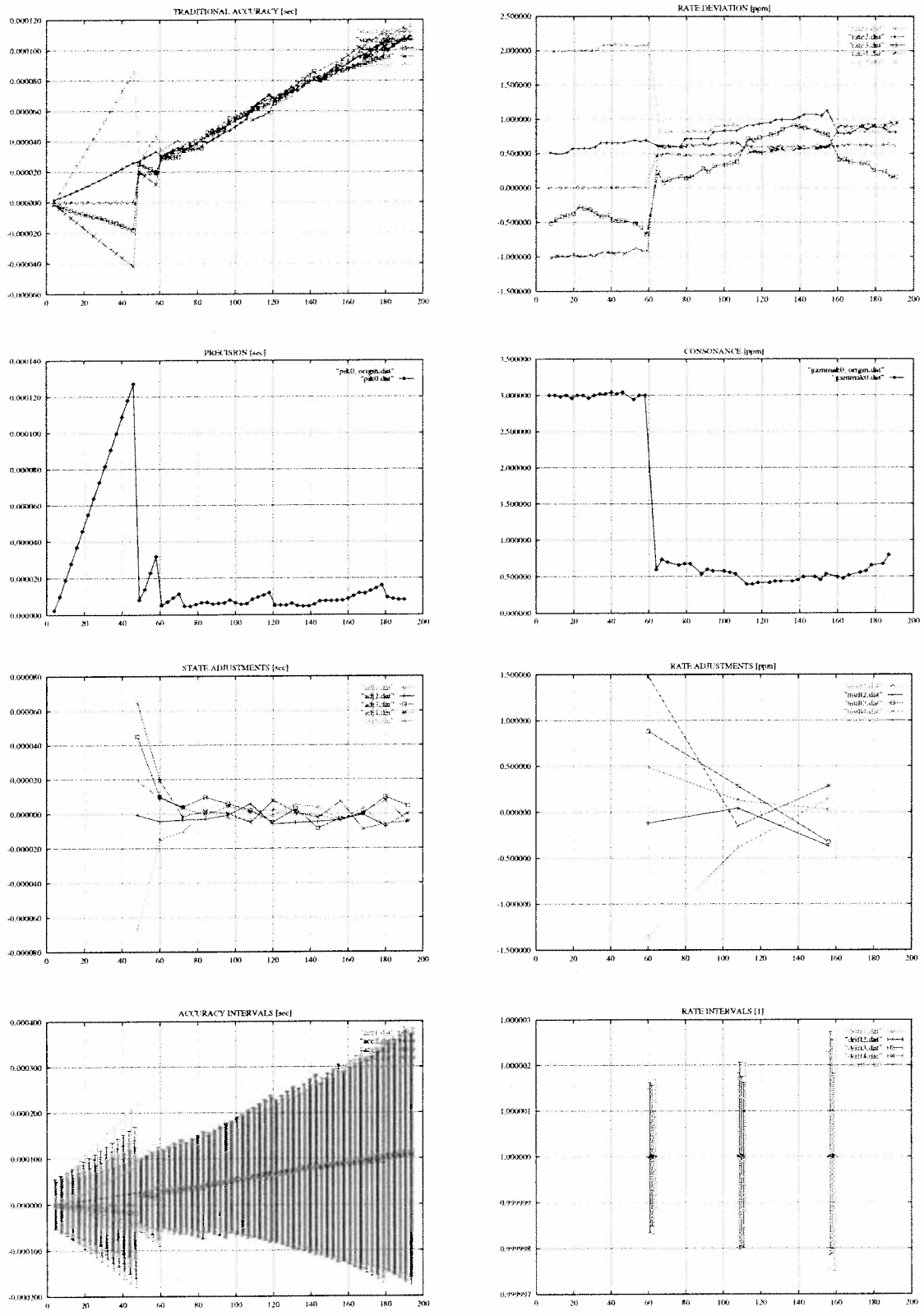


Figure 6: Plots of a Simulation Run

Conclusions

We decomposed our OP algorithm designed for clock state&rate synchronization to make it amenable for an implementation in our simulation toolkit SimUTC. Furthermore, we extended the accompanying evaluation system for plotting time-dependent simulation data like traditional accuracy, precision, state adjustment, accuracy intervals, rate deviation, consonance, rate adjustment, and rate intervals. Due to the complexity of algorithm OP and the diversity of interfaces many software engineering problems needed to be solved. The advantage of having documented the implementation process of algorithm OP are as follows: it helps to understand the algorithm itself, other synchronization algorithms can be emulated by special parameter settings, and the C++ code can be reused for further algorithms.

Acknowledgments

We would like to acknowledge Bettina Weiss for helping us with the SimUTC toolkit and for improving the C++ code.

References

[Sch97a] U. Schmid. Interval-based clock synchronization with optimal precision. Technical Report 183/1-78, Department of Automation, Vienna University of Technology, December 1997.

[Sch97b] K. Schossmaier. An interval-based framework for clock rate synchronization. In *Proceedings of the 16th Annual ACM Symposium on Principles on Distributed Computing (PODC)*, pages 169–178, Santa Barbara, CA, 21–24 August 1997.

[Sch98] K. Schossmaier. *Interval-based Clock State and Rate Synchronization.* PhD thesis, Faculty of Technical and Natural Sciences, Vienna University of Technology, September 1998.

[SS97] U. Schmid and K. Schossmaier. Interval-based clock synchronization. *Real-Time Systems*, 12(2):173–228, March 1997.

[Wei97] B. Weiss. *Simulation Environment for Clock Synchronization.* Diploma thesis, Department of Automation, Vienna University of Technology, June 1997.

[Wei99] B. Weiss. Simulation environment for clock synchronization. Technical Report 183/1-88, Department of Automation, Vienna University of Technology, February 1999.

Contents

Appendix (C++ Listings)

A SimUTC/Sources/Includes/algrate.hpp

```
//
// algrate.hpp
//
// declaration of the "clock rate synchronization algorithm"
//
// created: 07.10.98, Klaus Schossmaier
//
// last change: 20.11.98, Klaus Schossmaier
//
// last revision: 29.01.99
//
// RCS version control information:
//     $Revision: 1.1 $$State: Exp $$Date: 1999/02/03 16:10:06 $
//


#ifndef _ALGRATE_HPP_
#define _ALGRATE_HPP_


#include "algbase.hpp"
#include "alggen.hpp"
#include "database.hpp"
#include "time.hpp"


// additions for relative rate measurement
struct SRateTime : SRate
{
    CNtpTime localTime;
    CNtpTime remoteTime;
    double ref;
};


//****************************************************************************
// class CRate: provides the Rate algorithm
//****************************************************************************

class CRate : public ACClockSyncAlgorithm
{
public:

    //--------------------------------------------------------------------------
    //--- constructor, destructor
    //--------------------------------------------------------------------------

    CRate (CClockSync *cs);
    // assert: cs != NULL


    //--------------------------------------------------------------------------
    //--- compute and set the new clock interval
    //--------------------------------------------------------------------------

    virtual EBool ConvergenceFunction ();
    // returns True if a new interval could be computed (depends on the
    // number of nodes in the system and the maximum number of faulty
    // nodes); in that case 'newTime' is set to the new clock value; if
    // no valid interval could be computed, False is returned and 'newTime'
    // is not changed


    //--------------------------------------------------------------------------
    //--- add time intervals
```

```
    //-------------------------------------------------------------------------

    virtual EBool StoreTime (const STimestamp& time, double resync,
        SINT32 networkID, const CAddr& netAddress, int area, EBool gps);
    // if CS:resync is True, the time is not stored; so the algorithm
    // only starts working after resync is False; then the time is
    // passed on to StoreTime() of the base class


    //-------------------------------------------------------------------------
    //--- add transmission delay intervals
    //-------------------------------------------------------------------------

    virtual EBool StoreDelay (double localDelay, double remoteDelay,
        const SRate& remoteRate, SINT32 networkID, const CAddr& netAddress);
    // counts to N before setting CS::resync to False, calls StoreDelay
    // of base class to process delays


    //-------------------------------------------------------------------------
    //--- add rate intervals
    //-------------------------------------------------------------------------

    virtual EBool StoreRate (const SRate& remoteRate,
        const CNtpTime& localTime, const CNtpTime& remoteTime,
        SINT32 networkID, const CAddr& netAddress, int area);
    // called when rate intervals are arriving my node
    // do the preprocessing and fill the database with remote rate intervals


    //-------------------------------------------------------------------------
    //--- manage clock changes
    //-------------------------------------------------------------------------

    virtual void ClockChanged();
    // computes initialization data (clock deterioration, accuracy) for
    // the clock, and some clock dependent parameters needed by the algorithm


    //-------------------------------------------------------------------------
    //--- creation function of class CRate
    //-------------------------------------------------------------------------

    static ACClockSyncAlgorithm *CreateRate (CClockSync *cs);
    // creates and returns a new object of type CRate


protected:

    //-------------------------------------------------------------------------
    //--- set members that are modified during reset
    //-------------------------------------------------------------------------

    virtual void Init ();
    // sets cntDelay=0, resync. status to True, calls Init() of base class


private:

    int cntDelay;
    // number of delay measurements

    SINT32 nodeID;
    int area;
    // for identification

    double piHMinus;
    double piHPlus;
    // maximal precision interval
```

```
    double piOwnMinus;
    double piOwnPlus;
    // own precision interval

    double ups;
    // maximal state correction

    double sigma;
    // oscillator stability

    double gammaHMinus;
    double gammaHPlus;
    // maximal consonance interval

    double gammaOwnMinus;
    double gammaOwnPlus;
    // own consonance interval

    double thetaMax;
    // maximal rate correction

    EBool rateRound;
    // indicates if this round is also for rate synchronization
    // assumes that all nodes are in synch!

#ifdef PUBLIC_IN_TEST
public:
#endif

    //-------------------------------------------------------------------------
    //--- state algorithm
    //-------------------------------------------------------------------------

    EBool State (CNtpTime& point, double& alpha_minus, double& alpha_plus,
        SINT32 networkID);
    // computes the new clock interval for the network; returns True if the
    // computation was successful, False otherwise

    //-------------------------------------------------------------------------
    //--- rate algorithm
    //-------------------------------------------------------------------------

    EBool Rate (double& multStep, double& theta_minus, double& theta_plus,
        SINT32 networkID);
    // compute the new coupling factor (i.e. register STEP)
    // returns True if the computation was successful, False otherwise
};

#endif
```

B SimUTC/Sources/algrate.cpp

```
//
// algrate.cpp
//
// implementation of the "clock rate synchronization algorithm"
//
// created: 07.10.98, Klaus Schossmaier
//
// last change: 23.11.98, Klaus Schossmaier
//
// last revision: 04.02.99
//
// RCS version control information:
//     $Revision: 1.3 $$State: Exp $$Date: 1999/02/05 16:03:20 $
//

#include <stream.h>
#include <string.h>   // for memmove()

#include "address.hpp"
#include "addrman.hpp"
#include "alggen.hpp"
#include "algrate.hpp"
#include "clockifc.hpp"
#include "constant.hpp"
#include "database.hpp"
#include "error.hpp"
#include "time.hpp"


// enable terminal output during a run
#define RUN_SETTING
//#define RUN_RATE

// enable output files for gnuplot
//#define GNU_ACC
//#define GNU_PRE
//#define GNU_DRIFT
//#define GNU_CONS

// file names
#define NAME_ACC   "Evalsys/gnuplot/acc.dat"
#define NAME_CONS  "Evalsys/gnuplot/cons.dat"
#define NAME_DRIFT "Evalsys/gnuplot/drift.dat"
#define NAME_PRE   "Evalsys/gnuplot/pre.dat"

// provide some macros for closing these files
#ifdef GNU_ACC
  #define CloseAcc() (Facc.close())
#else
  #define CloseAcc() ((void)0)
#endif

#ifdef GNU_PRE
  #define ClosePre() (Fpre.close())
#else
  #define ClosePre() ((void)0)
#endif

#ifdef GNU_CONS
  #define CloseCons() (Fcons.close())
#else
  #define CloseCons() ((void)0)
#endif

#ifdef GNU_DRIFT
```

```
  #define CloseDrift() (Fdrift.close())
#else
  #define CloseDrift() ((void)0)
#endif


//***************************************************************************
// module global constants
//***************************************************************************

// --- default algorithmic parameters
// when undefined they are computed
#define PI_H        220e-6
#define PI_0        220e-6
#define UPSILON_MAX 300e-6
#define SIGMA_MAX   0.01e-6 // 0.01e-6
#define GAMMA_H     4e-6
#define GAMMA_0     4e-6
#define THETA_MAX   10e-6

// --- rate interval boundaries
#define RATE_MAX 2.0
#define RATE_MIN 0.0

// ---- initial accuracies of the clocks
// a delayed start could render them as not correct!
const double accPlusInitial  = 50e-6;
const double accMinusInitial = 50e-6;

// --- initial rate drifts of the clocks
// a delayed start could render them as not correct!
const double thetaPlusInitial  = 2e-6;
const double thetaMinusInitial = 2e-6;

// --- initial values for EpsilonPlus/Minus
// for the computation of the precision intervals
// they can be refined after some delay measurements
const double epsilonPlusInitial  = 10e-6;
const double epsilonMinusInitial = 10e-6;

// --- number of delay measurements
// before the algorithm starts using the delay data in StoreTime()
const int StartSynchronization = 14;

extern char *ftoa (double f, int width, char *string);  // from ftoa.cpp
// returns 'f' as a string with at least 'width' characters in %e format;
// stores the string in 'string', which must be big enough to hold the
// converted string; if 'string' is NULL, NULL is returned


//***************************************************************************
// module global functions
//***************************************************************************

static INT16 ThetaToLambda (double theta, double fnom);
// convert computed rate drift for a LAMBDA register

static void PlotInterval (ostream& dataStream, double x,
                          const SAsymInterval& In);
// write an (asym)interval into an .dat file for GNUPLOT


//***************************************************************************
// class CRate: features the Rate algorithm
//***************************************************************************

//---------------------------------------------------------------------------
//--- constructor, destructor
//---------------------------------------------------------------------------
```

```
// assert: cs != NULL (done by the ctor of ACClockSyncAlgorithm)
// for initialization beware the order:
// 1st the deterioration, 2nd the accuracy
CRate::CRate (CClockSync *cs) : ACClockSyncAlgorithm (cs)
{
    cntDelay = 0;
    // no delay measurements

    SetResynchronizationStatus (True);
    // own time is not valid

    cout.setf (ios::fixed, ios::floatfield);

    rateRound = False;
    // initial no rate round

    GetOwnID (nodeID, area);
    // determine the ID and store it in the member data

    ClockChanged();  // initialize clock dependent data
}


//---------------------------------------------------------------------------
//--- add time intervals
//---------------------------------------------------------------------------

// if CS:resync is True, the time is not stored; so the algorithm
// only starts working after resync is False; then the time is
// passed on to StoreTime() of the base class
EBool CRate::StoreTime (const STimestamp& time, double resync,
        SINT32 networkID, const CAddr& netAddress, int area, EBool gps)
{
    if (GetResynchronizationStatus ())
        return False;
    // do not store the time as long as CS is resynchronizing

    return ACClockSyncAlgorithm::StoreTime (time, resync, networkID,
        netAddress, area, gps);
}


//---------------------------------------------------------------------------
//--- add transmission delay intervals
//---------------------------------------------------------------------------

// counts to N before setting CS::resync to False, calls StoreDelay
// of base class to process delays
EBool CRate::StoreDelay (double localDelay, double remoteDelay,
     const SRate& remoteRate, SINT32 networkID, const CAddr& netAddress)
{
    if (cntDelay >= 0) cntDelay++;

    if (cntDelay == StartSynchronization)
    {
        SetResynchronizationStatus (False);
        // the own timestamp is valid

        cntDelay = -1;  // stop counting the delay measurement rounds
    }
    cntDelay++;

    if (cntDelay == StartSynchronization)
    {
        SetResynchronizationStatus (False);
    }
    // the own timestamp is valid
```

```
    return ACClockSyncAlgorithm::StoreDelay (localDelay, remoteDelay,
        remoteRate, networkID, netAddress);
}


//---------------------------------------------------------------------------
//--- add rate intervals
//---------------------------------------------------------------------------

// called when rate intervals are arriving my node
// do the preprocessing and fill the database with remote rate intervals
EBool CRate::StoreRate (const SRate& remoteRate, const CNtpTime& localTime,
    const CNtpTime& remoteTime, SINT32 networkID, const CAddr& netAddress,
    int remoteArea)
{
    SINT32 remoteID;           // nodeID of peer node
    SRateTime *rateData;       // pointer to the rate interval
    SRate evalRate;            // rate interval for EvalSys
    double deltaRemoteTime;    // expired time on the remote node
    double deltaLocalTime;     // expired time on the local node
    SAsymInterval quotientInt; // quotient rate interval
    SAsymInterval rateInt;     // input rate interval
    double lengthInt;          // interval length, half side
    SDelay delay;              // transmission delay parameters

    database->GetAssociation (networkID, netAddress, &remoteID);
    // get ID of the remote node

    // --- compute the quotient rate intervals

    quotientInt.ref   = 1.0;
    quotientInt.left  = 1.0;
    quotientInt.right = 1.0;
    // initialize the quotient rate interval

    rateData = (SRateTime *) database->GetRate (networkID, netAddress,
        remoteArea);
    // get rate data from last rate round, if any

    if (rateData != NULL)   // there was a previous rate round
    {
        // --- compute the refpoint of the quotient rate interval

        deltaRemoteTime =
            NtpToSeconds (remoteTime) - NtpToSeconds (rateData->remoteTime);
        // expired logical time on remote clock

        deltaLocalTime =
            NtpToSeconds (localTime) - NtpToSeconds (rateData->localTime);
        // expired logical time on local clock

        quotientInt.ref = (deltaRemoteTime - remoteRate.u) /
            (deltaLocalTime - rate->u);
       // take the ratio

        Assert (quotientInt.ref > RATE_MIN);
        Assert (quotientInt.ref < RATE_MAX);
        // relative rate measurement not useful

        // --- compute the lengths of the quotient rate interval

        if (!database->GetDelay (delay, networkID, netAddress))
        {
            return False;
            // alternatively use initial values!
        }
        // get transmission delay uncertainty

        lengthInt = sigma * (deltaRemoteTime - remoteRate.u) +
```

```
        (delay.epsilonMinus + delay.epsilonPlus) /
        (deltaLocalTime - rate->u);
    // length of each side

    quotientInt.left  = quotientInt.ref - lengthInt;
    quotientInt.right = quotientInt.ref + lengthInt;
    // set the quotient rate interval symmetricaly

    Assert (quotientInt.left > RATE_MIN);
    Assert (quotientInt.right < RATE_MAX);
    // relative rate measurement not useful

    rateRound = True;
    // informs the algorithm that this is a rate round
    // but not the very first time
}
else
{
    rateData = new SRateTime;
    database->SetRate (rateData, networkID, netAddress, remoteArea);
    // when there is no former entry, put pointer into the database
}

// --- compute remote rate intervals
// note that the very first one is not used

if ((nodeID == remoteID) && (area == remoteArea))
{
    // this is just the local rate interval
    rateInt.ref   = 1.0;
    rateInt.left  = 1.0 - remoteRate.thetaMinus;
    rateInt.right = 1.0 + remoteRate.thetaPlus;
}
else
{
    rateInt.ref = quotientInt.ref;
    // take over the reference point

    rateInt.left = quotientInt.left * (1 - remoteRate.thetaMinus) -
        2 * sigma * ( GetDelayCompensation () +
            GetComputationDelayCompensation () );
    // left edge of remote rate interval

    rateInt.right = quotientInt.right * (1 + remoteRate.thetaPlus) +
        2 * sigma * ( GetDelayCompensation () +
            GetComputationDelayCompensation () );
    // right edge of remote rate interval
}

Assert (rateInt.left  > RATE_MIN);
Assert (rateInt.right < RATE_MAX);
// relative rate measurement or remote rate interval not useful

// --- save the current rate data in the database; has two functions:
// 1) remember timestamps for the next rate synchronization
// 2) hand over rate interval to the CV

rateData->thetaPlus = rateInt.right - rateInt.ref;
rateData->thetaMinus = rateInt.ref - rateInt.left;
rateData->u = remoteRate.u;
rateData->ref = rateInt.ref;
rateData->localTime = localTime;
rateData->remoteTime = remoteTime;
// compose the entry

evalRate.thetaPlus = rateData->thetaPlus;
evalRate.thetaMinus = rateData->thetaMinus;
evalRate.u = remoteRate.u;
ReportRate (True, evalRate, quotientInt.ref);
```

```
    // report rate interval to EvalSys, as input

    return True;
}


//------------------------------------------------------------------------
//--- manage clock changes
//------------------------------------------------------------------------

// computes initialization data (clock deterioration, accuracy) for
// the clock, and some clock dependent parameters needed by the algorithm
void CRate::ClockChanged()
{
    CClockInterface *clock;        // access to the clock
    double fnom;                   // oscillator frequency
    STimestamp state;              // current clock state
    SExtendedTimestamp state0;     // clock state with desired accuray
    double pi0Minus, pi0Plus;      // initial precisions
    INT16 lambdaMinus, lambdaPlus; // deteriorations of accuracies


    ACClockSyncAlgorithm::ClockChanged();  // initialize the local rate

    clock = GetClock();
    if (clock == NULL) return;
    // the clock is not available, so we cannot initialize it
    // note: we could report that the clock is NULL, since the user may not
    // have intended this; but the report might come into the way of object
    // destruction


    #ifdef RUN_SETTING
        cout << "----- Setting of node " << nodeID << " -----" << endl;
    #endif

    // --- initial rate interval R_init = [thetaMinusInitial,thetaPlusInitial]

    rate->thetaMinus = thetaMinusInitial;
    rate->thetaPlus  = thetaPlusInitial;
    // set the initial local rate interval

    #ifdef RUN_SETTING
        cout << "R_init   = ["
             << rate->thetaMinus * 1e6 << ", "
             << rate->thetaPlus * 1e6 << "] ppm" << endl;
    #endif

    fnom = clock->GetFrequency ();
    lambdaMinus = ThetaToLambda (thetaMinusInitial, fnom);
    lambdaPlus  = ThetaToLambda (thetaPlusInitial, fnom);
    clock->SetLambdapure (lambdaMinus, lambdaPlus);
    // set the initial deterioration according to R_init

    // --- initial accuracy interval A_init = [accMinusInitial,accPlusInitial]

    state0 = IntervalToClock (0, accMinusInitial, accPlusInitial);
    // compute the accuracy values, reference point is unused

    clock->GetTimestamp (state);
    // get the current clock state

    state.acc = state0.acc;
    clock->SetTimestamp (state);
    // set the initial accuracies

    #ifdef RUN_SETTING
        cout << "A_init   = ["
             << accMinusInitial << ", " << accPlusInitial << "] s" << endl;
```

```
#endif

// --- precision interval pi_0 = [piOMinus,piOPlus]

piOMinus = (epsilonPlusInitial + epsilonMinusInitial) +
      4 * clock->GetGranularity () +
      0.5 * (GetRoundPeriod () + GetDelayCompensation () +
             2 * GetComputationDelayCompensation ())
          * (clock->GetRhoPlus () + clock->GetRhoMinus ()) +
      0.5 * GetRoundPeriod ()
          * (clock->GetRhoPlus () - clock->GetRhoMinus ());

piOPlus = (epsilonPlusInitial + epsilonMinusInitial) +
      4 * clock->GetGranularity () +
      0.5 * (GetRoundPeriod () + GetDelayCompensation () +
             2 * GetComputationDelayCompensation ())
          * (clock->GetRhoPlus () + clock->GetRhoMinus ()) -
      0.5 * GetRoundPeriod ()
          * (clock->GetRhoPlus () - clock->GetRhoMinus ());

// --- precision interval pi^H = [piHMinus,piHPlus]

#ifdef PI_H
    piHMinus = piHPlus = PI_H;
#else
    piHMinus = piOMinus + epsilonMinusInitial +
        3 * clock->GetGranularity () +
        (GetRoundPeriod () + GetComputationDelayCompensation ())
        * clock->GetRhoMinus ();

    piHPlus = piOPlus + epsilonPlusInitial +
        2 * clock->GetGranularity () +
        (GetRoundPeriod () + GetComputationDelayCompensation ())
        * clock->GetRhoPlus ();
#endif

#ifdef RUN_SETTING
    cout << "piH      = ["
         << piHMinus << ", " << piHPlus << "] s" << endl;
#endif

// --- precision interval pi^o = [piOwnMinus,piOwnPlus]

#ifdef PI_O
    piOwnMinus = piOwnPlus = PI_O;
#else
    piOwnMinus = piOMinus + clock->GetGranularity () +
          GetRoundPeriod () * clock->GetRhoMinus ();

    piOwnPlus = piOPlus + clock->GetGranularity () +
          GetRoundPeriod () * clock->GetRhoPlus ();
#endif

#ifdef RUN_SETTING
    cout << "piOwn    = ["
         << piOwnMinus << ", " << piOwnPlus << "] s" << endl;
#endif

// --- maximal state correction Upsilon_max:

#ifdef UPSILON_MAX
    ups = UPSILON_MAX;
#else
    ups = 2 * clock->GetGranularity () +
        GetRoundPeriod () * (clock->GetRhoPlus () + clock->GetRhoMinus ());
#endif

#ifdef RUN_SETTING
    cout << "ups      = " << ups << " s " << endl;
```

```
    #endif

    // --- oscillator stability

    sigma = SIGMA_MAX;  // no calculation, no measurement

    #ifdef RUN_SETTING
        cout << "sigma    = " << sigma * 1e6 << " ppm/s " << endl;
    #endif

    // --- consonance interval gamma_H = [gammaHMinus,gammaHPlus]

    #ifdef GAMMA_H
        gammaHMinus = gammaHPlus = GAMMA_H;
    #else
        gammaHMinus = gammaHPlus = sigma * ( 3 * GetRatePeriod () +
            10 * GetDelayCompensation () +
            6 * GetComputationDelayCompensation () ) +
            3 * (epsilonPlusInitial + epsilonMinusInitial) /
                ( GetRatePeriod () - GetDelayCompensation () -
                epsilonPlusInitial - epsilonMinusInitial );
    #endif

    #ifdef RUN_SETTING
        cout << "gammaH   = ["
             << gammaHMinus * 1e6 << ", " << gammaHPlus * 1e6 << "] ppm "
             << endl;
    #endif

    // --- consonance interval gamma_O = [gammaOwnMinus,gammaOwnPlus]

    #ifdef GAMMA_O
        gammaOwnMinus = gammaOwnPlus = GAMMA_O;
    #else
        gammaOwnMinus = gammaOwnPlus = sigma * ( 3 * GetRatePeriod () +
            7 * GetDelayCompensation () +
            4 * GetComputationDelayCompensation () ) +
            2 * (epsilonPlusInitial + epsilonMinusInitial) /
                ( GetRatePeriod () - GetDelayCompensation () -
                epsilonPlusInitial - epsilonMinusInitial );
    #endif

    #ifdef RUN_SETTING
        cout << "gammaOwn = ["
             << gammaOwnMinus * 1e6 << ", " << gammaOwnPlus * 1e6 << "] ppm "
             << endl;
    #endif

    // --- maximal rate correction Theta_max

    #ifdef THETA_MAX
        thetaMax = THETA_MAX;
    #else
        thetaMax = 2 * sigma * GetRatePeriod ();
    #endif

    #ifdef RUN_SETTING
        cout << "thetaMax = " << thetaMax * 1e6 << " ppm " << endl;
    #endif

    cout << endl;
}


//------------------------------------------------------------------------
//--- create function of class CRate
//------------------------------------------------------------------------

// creates and returns a new object of type CRate
```

```
ACClockSyncAlgorithm *CRate::CreateRate (CClockSync *cs)
{
    return new CRate (cs);
}


//---------------------------------------------------------------------------
//--- compute and set the new clock interval and new rate interval
//---------------------------------------------------------------------------

// first synchronize the rate and then the state, otherwise
// the accumulation of the state adjustments is not correct
EBool CRate::ConvergenceFunction ()
{
    CClockInterface *clock;          // access to the clock
    double fnom;                     // oscillator frequency
    CNtpTime point;                  // new reference point
    double alpha_minus, alpha_plus;  // new accuracies
    double theta_minus, theta_plus;  // new rate drifts
    double multStep;                 // factor for rate adjustment
    double stepNew;                  // new coupling factor
    INT32 step;                      // UTCSU step register
    INT16 lambdaMinus, lambdaPlus;   // UTCSU deterioration registers
    STimeInterval evalState;         // state interval for EvalSys
    SRate evalRate;                  // rate interval for EvalSys

    timeValid = False;
    // first of all mark the time as invalid

    if (GetAssociatedNetwork() == NoAccount)
        return False;
    // no associated network

    clock = GetClock();
    Assert (clock != NULL);
    // clock must exist

    // --- synchronize the clock rate

    if (rateRound) // this a rate round as well
    {
        if (!Rate (multStep, theta_minus, theta_plus, GetAssociatedNetwork()))
            return False;
        // Rate() function failed

        step = clock->GetSteppure();
        // the current coupling factor

        stepNew = step * multStep;
        // new coupling factor by multiplication

        Assert (stepNew <= 0xFFFFFFFF);
        // coupling factor is not useful

        clock->SetSteppure ((INT32)(stepNew));
        // change the clock rate

        rate->thetaMinus = theta_minus;
        rate->thetaPlus  = theta_plus;
        // set new local rate interval, without deterioration

        rateRound = False;
        // next round is not for rate

        evalRate.thetaPlus = theta_plus;
        evalRate.thetaMinus = theta_minus;
        evalRate.u = 0.0;
        ReportRate (False, evalRate, multStep);
        // report new rate interval to EvalSys, as output
```

```
    } // if

    rate->thetaMinus += (sigma * GetRoundPeriod ());
    rate->thetaPlus  += (sigma * GetRoundPeriod ());
    // deteriorate local rate interval for the next state round

    fnom = clock->GetFrequency ();
    lambdaMinus = ThetaToLambda (rate->thetaMinus, fnom);
    lambdaPlus  = ThetaToLambda (rate->thetaPlus, fnom);
    clock->SetLambdapure (lambdaMinus, lambdaPlus);
    // set the new accuracy interval deteriorations

    // --- synchronize the clock state

    if (!State (point, alpha_minus, alpha_plus, GetAssociatedNetwork()))
        return False;
    // State() function failed

    *newClockValue = IntervalToClock (point, alpha_minus, alpha_plus);
    // set new clock interval

    evalState.refpoint = point;
    evalState.low  = point - CNtpTime (alpha_minus);
    evalState.high = point + CNtpTime (alpha_plus);
    ReportInterval (False, evalState);
    // report new state interval to EvalSys, as output

    timeValid = True;
    // the new timestamp is valid

    return True;
}


//-----------------------------------------------------------------------------
//--- state algorithm
//-----------------------------------------------------------------------------

// computes the new clock interval for the network;
// returns True if the computation was successful, False otherwise
EBool CRate::State (CNtpTime& point, double& alpha_minus, double& alpha_plus,
    SINT32 networkID)
{
    STimeInterval data; // read out the database of state intervals
    EBool entryExists;  // is there a state interval in the database
    INT32 sizeState;    // number of state intervals in the database
    int i;
    CAddr netAddress;
    SAsymInterval *accuracyIntervals;  // set of accuracy intervals
    SAsymInterval *precisionIntervals; // set of precision intervals
    SAsymInterval preIntMarz;          // precision interval after Marzullo
    SAsymInterval preIntResult;        // precision interval towards final
    SAsymInterval accIntMarz;          // accuracy interval after Marzullo
    SAsymInterval accIntResult;        // accuracy interval towards final
    double adjustment;
    filebuf Facc, Fpre; // write intervals to .dat files for gnuplot

    // --- enough state intervals

    if (numberOfNodes <= 3*numberOfFaultyNodes) // n > 3f is needed for OP
    {
        database->DeleteAll (CApplicationDatabase::Time, networkID);
        return False;
    }

    sizeState = database->Cardinality (CApplicationDatabase::Time, networkID);
    if (sizeState <= 2*numberOfFaultyNodes)
    // need at least 2*f + 1 timestamps
    {
```

```
        database->DeleteAll (CApplicationDatabase::Time, networkID);
        return False;
    }

    accuracyIntervals = new SAsymInterval[sizeState];
    precisionIntervals = new SAsymInterval[sizeState];
    // these should be the input state intervals represented in doubles

    // --- get own accuracy/precision interval

    if (!database->GetAssociation (networkID, nodeID, netAddress))
    {
        delete[] accuracyIntervals;
        delete[] precisionIntervals;
        database->DeleteAll (CApplicationDatabase::Time, networkID);
        return False;
    }
    // obtain netAddress

    if (!database->RemoveTime (data, networkID, netAddress, area))
    {
        delete[] accuracyIntervals;
        delete[] precisionIntervals;
        database->DeleteAll (CApplicationDatabase::Time, networkID);
        return False;
    }
    // obtain own accuracy interval

    accuracyIntervals[0].left  = NtpToSeconds (data.low);
    accuracyIntervals[0].ref   = NtpToSeconds (data.refpoint);
    accuracyIntervals[0].right = NtpToSeconds (data.high);
    // own accuracy interval represented in doubles, index = 0

    precisionIntervals[0].left  = accuracyIntervals[0].ref - piOwnMinus;
    precisionIntervals[0].ref   = accuracyIntervals[0].ref;
    precisionIntervals[0].right = accuracyIntervals[0].ref + piOwnPlus;
    // own precision interval represented in doubles, index = 0

    // --- get remote accuracy/precision intervals

    for (entryExists = database->GetFirstTime (data, networkID), i = 1;
         entryExists;
         entryExists = database->GetNextTime (data, networkID), i++)
    {
         accuracyIntervals[i].left  = NtpToSeconds (data.low);
         accuracyIntervals[i].ref   = NtpToSeconds (data.refpoint);
         accuracyIntervals[i].right = NtpToSeconds (data.high);
         // preprocessed accuracy interval

         precisionIntervals[i].left  = accuracyIntervals[i].ref - piHMinus;
         precisionIntervals[i].ref   = accuracyIntervals[i].ref;
         precisionIntervals[i].right = accuracyIntervals[i].ref + piHPlus;
         // computed precision interval
    } // for

    // now we have the accuracy and precision intervals at hand
    // in accuracyIntervals[] and precisionIntervals[] respectively,
    // where in [0] are the own ones

#ifdef GNU_PRE
    if (Fpre.open(NAME_PRE,output) == 0)
    {
        delete[] accuracyIntervals;
        delete[] precisionIntervals;
        database->DeleteAll (CApplicationDatabase::Time, networkID);
        return False;
    }
    ostream pre(&Fpre);
    for(i=0;i<sizeState;i++)
```

```
            PlotInterval (pre, i, precisionIntervals[i]);
    #endif

        // --- apply convergence function OP upon the precision intervals

        if (!ApplyMarzullo (precisionIntervals, preIntMarz, sizeState,
            numberOfFaultyNodes))
        {
            delete[] accuracyIntervals;
            delete[] precisionIntervals;
            database->DeleteAll (CApplicationDatabase::Time, networkID);
            ClosePre();
            return False;
        }
        // Marzullo's function upon associated precision intervals

    #ifdef GNU_PRE
        PlotInterval (pre, 6, preIntMarz);
    #endif

        if (!Intersect (preIntMarz, precisionIntervals[0], preIntResult))
        {
            delete[] accuracyIntervals;
            delete[] precisionIntervals;
            database->DeleteAll (CApplicationDatabase::Time, networkID);
            ClosePre();
            return False;
        }
        // intersection with local precision interval

        preIntResult.ref =
            (piOwnMinus * preIntResult.right + piOwnPlus * preIntResult.left) /
            (piOwnMinus + piOwnPlus);
        // candidat reference point

    #ifdef GNU_PRE
        PlotInterval (pre, 7, preIntResult);
    #endif

        adjustment = preIntResult.ref - precisionIntervals[0].ref;

        if (adjustment > ups)
            preIntResult.ref = precisionIntervals[0].ref + ups;
        if (adjustment < - ups)
            preIntResult.ref = precisionIntervals[0].ref - ups;
        // new reference point

        rate->u += (preIntResult.ref - precisionIntervals[0].ref);
        // accumulate state adjustment for relative rate measurement

    #ifdef GNU_PRE
        PlotInterval (pre, 8, preIntResult);
    #endif

        // --- apply convergence function OP upon the accuracy intervals

    #ifdef GNU_ACC
        if (Facc.open(NAME_ACC,output) == 0)
        {
            delete[] accuracyIntervals;
            delete[] precisionIntervals;
            database->DeleteAll (CApplicationDatabase::Time, networkID);
            ClosePre();
            return False;
        }
        ostream acc(&Facc);
        for(i=0;i<sizeState;i++)
            PlotInterval (acc, i, accuracyIntervals[i]);
    #endif
```

```
    if (!ApplyMarzullo (accuracyIntervals, accIntMarz, sizeState,
        numberOfFaultyNodes))
    {
        delete[] accuracyIntervals;
        delete[] precisionIntervals;
        database->DeleteAll (CApplicationDatabase::Time, networkID);
        ClosePre();
        CloseAcc();
        return False;
    }
    // Marzullo's function upon accuracy intervals

#ifdef GNU_ACC
    PlotInterval (acc, 6, accIntMarz);
#endif

    if (!Intersect (accIntMarz, accuracyIntervals[0], accIntResult))
    {
        delete[] accuracyIntervals;
        delete[] precisionIntervals;
        database->DeleteAll (CApplicationDatabase::Time, networkID);
        ClosePre();
        CloseAcc();
        return False;
    }
    // intersection with own accuracy interval

#ifdef GNU_ACC
    PlotInterval (acc, 7, accIntResult);
#endif

    if (preIntResult.ref > accIntResult.right)
        accIntResult.right = preIntResult.ref;
    if (preIntResult.ref < accIntResult.left)
        accIntResult.left = preIntResult.ref;
    // new edges of accuracy interval

#ifdef GNU_ACC
    PlotInterval (acc, 8, accIntResult);
#endif

    // --- compute the resulting values

    SecondsToNtp (preIntResult.ref, point.ms, point.ts, point.us);
    // hand back the new reference point, from precision intervals

    alpha_minus = preIntResult.ref - accIntResult.left;
    Assert (alpha_minus >= 0);
    // hand back the new left edge

    alpha_plus = accIntResult.right - preIntResult.ref;
    Assert (alpha_plus >= 0);
    // hand back the new right edge

    delete[] accuracyIntervals;
    delete[] precisionIntervals;
    database->DeleteAll (CApplicationDatabase::Time, networkID);
    // the data is not needed anymore

    ClosePre();
    CloseAcc();
    // close the debug files if necessary

    return True;
}


//------------------------------------------------------------------------
```

```
//--- rate algorithm
//-------------------------------------------------------------------------

// compute the new coupling factor (for STEP register) and
// local rate interval (for LAMBDA registers)
// returns True if the computation was successful, False otherwise
EBool CRate::Rate (double& multStep, double& theta_minus, double& theta_plus,
    SINT32 networkID)
{
    SRate *rateEntryExists; // pointer to rate intervals in the database
    SRate *rateEntryOwn;    // pointer to own rate interval in the database
    INT32 sizeRate;         // number of rate intervals in the database
    int i;
    CAddr netAddress;
    SAsymInterval *rateIntervals;       // set of rate intervals
    SAsymInterval *consonanceIntervals; // set of consonance intervals
    SAsymInterval consIntMarz;          // consonance intervals after Marzullo
    SAsymInterval consIntResult;        // consonance intervals towards final
    SAsymInterval rateIntMarz;          // rate intervals after Marzullo
    SAsymInterval rateIntResult;        // rate interval towards final
    filebuf Fcons, Fdrift; // write intervals to .dat files for GNUPLOT

   rate->u = 0;
   // clear state adjustments for next relative rate measurement

   // --- enough rate intervals

   if (numberOfNodes <= 3*numberOfFaultyNodes) // n > 3f is needed for OP
   {
       // database->DeleteAll (CApplicationDatabase::Rate, networkID);
       return False;
   }

   sizeRate = database->Cardinality (CApplicationDatabase::Rate, networkID);
   if (sizeRate <= 2*numberOfFaultyNodes)
   // need at least 2*f + 1 timestamps
   {
       // database->DeleteAll (CApplicationDatabase::Rate, networkID);
       return False;
   }

   rateIntervals = new SAsymInterval[sizeRate];
   consonanceIntervals = new SAsymInterval[sizeRate];
   // these should be the input rate intervals represented in doubles

   // --- get own rate/consonance interval

   if (!database->GetAssociation (networkID, nodeID, netAddress))
   {
       delete[] rateIntervals;
       delete[] consonanceIntervals;
       // database->DeleteAll (CApplicationDatabase::Rate, networkID);
       return False;
   }
   // obtain netAddress

   rateEntryOwn = database->RemoveRate (networkID, netAddress, area);
   if (rateEntryOwn == NULL)
   {
       delete[] rateIntervals;
       delete[] consonanceIntervals;
       // database->DeleteAll (CApplicationDatabase::Rate, networkID);
       return False;
   }
   // obtain own rate interval

   rateIntervals[0].ref   = ((SRateTime *)rateEntryOwn)->ref;
   rateIntervals[0].left  = rateIntervals[0].ref - rateEntryOwn->thetaMinus;
   rateIntervals[0].right = rateIntervals[0].ref + rateEntryOwn->thetaPlus;
```

```
    // own rate interval represented in  doubles, index = 0

    consonanceIntervals[0].left  = rateIntervals[0].ref - gammaOwnMinus;
    consonanceIntervals[0].ref   = rateIntervals[0].ref;
    consonanceIntervals[0].right = rateIntervals[0].ref + gammaOwnPlus;
    // own consonance interval represented in doubles, index = 0

    // --- get remote rate/consonance intervals

    for (rateEntryExists = database->GetFirstRate (networkID), i = 1;
         rateEntryExists != NULL;
         rateEntryExists = database->GetNextRate (networkID), i++)
    {
        rateIntervals[i].ref = ((SRateTime *)rateEntryExists)->ref;
        rateIntervals[i].left = rateIntervals[i].ref -
            rateEntryExists->thetaMinus;
        rateIntervals[i].right = rateIntervals[i].ref +
            rateEntryExists->thetaPlus;
        // preprocessed rate interval

        consonanceIntervals[i].left  = rateIntervals[i].ref - gammaHMinus;
        consonanceIntervals[i].ref   = rateIntervals[i].ref;
        consonanceIntervals[i].right = rateIntervals[i].ref + gammaHPlus;
        // computed consonance interval
    } // for

    database->SetRate (rateEntryOwn, networkID, netAddress, area);
    // put the own rate interval back into the database
    // necessary to carry out the relative rate measurement

    // now we have the rate and consonance intervals at hand
    // in rateIntervals[] and consonanceIntervals[] respectively,
    // where in [0] are the own ones

    // --- apply convergence function OP upon the consonance intervals

#ifdef GNU_CONS
    if (Fcons.open(NAME_CONS,output) == 0)
    {
        delete[] consonanceIntervals;
        delete[] rateIntervals;
        return False;
    }
    ostream cons(&Fcons);
    for(i=0;i<sizeRate;i++)
        PlotInterval (cons, i, consonanceIntervals[i]);
#endif

    if (!ApplyMarzullo (consonanceIntervals, consIntMarz, sizeRate,
                        numberOfFaultyNodes))
    {
        delete[] consonanceIntervals;
        delete[] rateIntervals;
        CloseCons();
        return False;
    }
    // Marzullo's function upon the consonance intervals

#ifdef GNU_CONS
    PlotInterval (cons, 6, consIntMarz);
#endif

    if (!Intersect (consIntMarz, consonanceIntervals[0], consIntResult))
    {
        delete[] consonanceIntervals;
        delete[] rateIntervals;
        CloseCons();
        return False;
    }
```

```
    // intersection with own precision interval

    consIntResult.ref =
        (gammaOwnMinus * consIntResult.right +
            gammaOwnPlus * consIntResult.left) /
        (gammaOwnMinus + gammaOwnPlus);
    // candidat reference point

#ifdef GNU_CONS
    PlotInterval (cons, 7, consIntResult);
#endif

    if ((consIntResult.ref - 1) > thetaMax)
        consIntResult.ref = 1.0 + thetaMax;
    if ((consIntResult.ref - 1) < - thetaMax)
        consIntResult.ref = 1.0 - thetaMax;
    // new reference point

#ifdef GNU_CONS
    PlotInterval (cons, 8, consIntResult);
#endif

    multStep = consIntResult.ref;
    // hand back the relative rate change of the local clock

    // --- apply convergence function OP upon the rate intervals

#ifdef GNU_DRIFT
    if (Fdrift.open(NAME_DRIFT,output) == 0)
    {
        delete[] consonanceIntervals;
        delete[] rateIntervals;
        CloseCons();
        return False;
    }
    ostream drift(&Fdrift);
    for(i=0;i<sizeRate;i++)
        PlotInterval (drift, i, rateIntervals[i]);
#endif

    if (!ApplyMarzullo (rateIntervals, rateIntMarz, sizeRate,
                        numberOfFaultyNodes))
    {
        delete[] consonanceIntervals;
        delete[] rateIntervals;
        CloseCons();
        CloseDrift();
        return False;
    }
    // Marzullo's function upon the rate intervals

#ifdef GNU_DRIFT
    PlotInterval (drift, 6, rateIntMarz);
#endif

    if (!Intersect (rateIntMarz, rateIntervals[0], rateIntResult))
    {
        delete[] consonanceIntervals;
        delete[] rateIntervals;
        CloseCons();
        CloseDrift();
        return False;
    }
    // intersection with own rate interval

#ifdef GNU_DRIFT
    PlotInterval (drift, 7, rateIntResult);
#endif
```

```
    if (consIntResult.ref > rateIntResult.right)
        rateIntResult.right = consIntResult.ref;
    if (consIntResult.ref < rateIntResult.left)
        rateIntResult.left = consIntResult.ref;
    // new edges of rate interval

#ifdef GNU_DRIFT
    PlotInterval (drift, 8, rateIntResult);
#endif

    // --- compute the resulting values

    Assert (consIntResult.ref > 0);

    theta_minus = (consIntResult.ref - rateIntResult.left)/consIntResult.ref;
    // normalize the negative rate drift with the consonance refpoint
    // no deterioration at here

    Assert (theta_minus >= 0);
    Assert (theta_minus <= 1);
    // negative rate drift is not useful

    theta_plus = (rateIntResult.right - consIntResult.ref)/consIntResult.ref;
    // normalize the positive rate drift with the consonance refpoint
    // no deterioration at here

    Assert (theta_plus >= 0);
    // positive rate drift is not useful

#ifdef RUN_RATE
    cout << "drift rates for node " << nodeID << ": "
         << theta_minus * 1e6 << ", " << theta_plus * 1e6<< endl;
#endif

    delete[] rateIntervals;
    delete[] consonanceIntervals;
    // the data is not needed anymore

    CloseCons();
    CloseDrift();
    // close the debug files if necessary

    return True;
}


//-----------------------------------------------------------------------------
//--- set members that are modified during reset
//-----------------------------------------------------------------------------

// sets cntDelay=0, resync. status to True, calls Init() of base class
void CRate::Init ()
{
    ACClockSyncAlgorithm::Init();

    cntDelay = 0;  // no delay measurements
    SetResynchronizationStatus (True);  // own time is not valid

    rateRound = False;  // initial no rate round
}



//*****************************************************************************
// module global functions
//*****************************************************************************

//-----------------------------------------------------------------------------
//--- convert computed rate drift for a LAMBDA register
```

```
//---------------------------------------------------------------------------

// for fnom = 25Mhz
// theta = 0.1e-6 yields 0x0009
// theta =   1e-6 yields 0x005A
// theta =  10e-6 yields 0x0384
// theta = 100e-6 yields 0x232F
// theta = 360e-6 yields 0x7EA9
static INT16 ThetaToLambda (double theta, double fnom)
{
    double lambda;
    CNtpTime lambdaRegister;

    Assert (theta > 0);
    Assert (fnom > 0);
    lambda = theta / fnom;
    // deterioration per tick, in double

    SecondsToNtp (lambda,
        lambdaRegister.ms, lambdaRegister.ts, lambdaRegister.us);
    // convert to binary where .us is (-25,-56)

    if (lambdaRegister.ms > 0)
        return 0x8FFF;
    if (lambdaRegister.ts > 0)
        return 0x8FFF;
    if ((lambdaRegister.us & 0xFFF00000) > 0)
        return 0x8FFF;
    // in case of an overflow hand back the possible maximum

    return (INT16)(lambdaRegister.us >> 5);
    // need to select SIGN(-37,-51), hence >>5
}


//---------------------------------------------------------------------------
//--- write an (asym)interval into an .dat file for GNUPLOT
//---------------------------------------------------------------------------

static void PlotInterval (ostream& dataStream, double x,
    const SAsymInterval& In)
{
    char val[40];

    ftoa(x,14,val);
    dataStream << val << " ";
    ftoa(In.ref,14,val);
    dataStream << val << " ";
    ftoa(In.left,14,val);
    dataStream << val << " ";
    ftoa(In.right,14,val);
    dataStream << val << endl;
}
```

C SimUTC/Sources/Includes/alggen.hpp

```
//
// alggen.hpp
//
// declaration of general matters for a "clock synchronization algorithm"
//
// created: 03.07.97, Klaus Schossmaier
//
// last change: 03.07.97, Klaus Schossmaier
//
// last revision: 21.01.99
//
// RCS version control information:
//     $Revision: 1.1 $$State: Exp $$Date: 1999/02/03 16:10:06 $
//


#ifndef _ALGGEN_HPP_
#define _ALGGEN_HPP_

#include "psos.h"

#include "bool.hpp"
#include "time.hpp"


//***************************************************************************
// structures
//***************************************************************************

// asymetrical interval
struct SAsymInterval
{
    double left;   // left edge
    double ref;    // reference point
    double right;  // right edge
};


//***************************************************************************
// constants
//***************************************************************************

// for accuray conversion
const INT32 Infinity = 0xFFFFFFFF;
// infinite accuracy; all 1 in (-8,-24)

const INT32 HighestAccuracyBit = 16;
// highest bit in timestamp that is covered by the accuracy, as seen from
// the lsb of timestamp (lsb = -24, highest bit = -8)

const INT32 HighestAccuracyValue = (1 << HighestAccuracyBit+1)-1;
// the highest value in timestamp that is covered by the accuracy

const int WidthOfChars = 14;
// used for function ftoa(), sets the number of used characters in %e format


//***************************************************************************
// functions
//***************************************************************************

extern EBool ApplyMarzullo (SAsymInterval* input, SAsymInterval& output,
                            INT32 size, int f);
// Marzullo function is applied on the input array 'input' which has
// size 'size', 'f' denotes the number of faulty nodes
```

```
// the result of the Marzullo function is returned in interval 'output'
// assert: input != NULL

extern void SortArray (double* unsortedArray, double* sortedArray, INT32 size);
// sorts the array 'unsortedArray' which has size 'size', returns the
// result in the array 'sortedArray' (which must be large enough to contain
// the sorted array)
// assert: unsortedArray != NULL, sortedArray != NULL, size > 0

extern EBool Intersect (const SAsymInterval& in1, const SAsymInterval& in2,
                        SAsymInterval& out);
// computes the intersection of intervals 'in1' and 'in2' and returns the
// result in 'out'; if the intersection exists, True is returned
// if the intersection is empty, False is returned, and in this case the
// contents of 'out' is not valid

extern SExtendedTimestamp IntervalToClock (CNtpTime point, double alpha_minus,
                                           double alpha_plus);
// converts the time 'point' with the accuracies 'alpha_minus' and
// 'alpha_plus' into a clock timestamp and returns this timestamp; if the
// accuracies exceed the allowed range, SExtendedTimestamp::acc is set
// to the value representing infinity

#endif
```

D SimUTC/Sources/alggen.cpp

```
//
// alggen.cpp
//
// contains constants and functions that are needed by several algorithms
//
// created: 19.01.99
//
// last change: 21.01.99
//
// RCS version control information:
//     $Revision: 1.1 $$State: Exp $$Date: 1999/02/04 14:29:21 $
//

#include <stdlib.h>  // for NULL
#include <string.h>  // for memmove()

#include "alggen.hpp"
#include "error.hpp"


//***************************************************************************
// functions
//***************************************************************************

//---------------------------------------------------------------------------
//--- Marzullo function
//---------------------------------------------------------------------------

EBool ApplyMarzullo (SAsymInterval* input, SAsymInterval& output, INT32 size,
                     int f)
{
    Assert (input != NULL);

    int i, j;
    int cut;
    int minCut;
    double sweep;
    double *sortedArray;
    double *unsortedArray;

    minCut = size - f;
    if (minCut < 0 ) return False;
    // number of required intersections

    sortedArray = new double[size];
    unsortedArray = new double[size];

    // --- determine the left edge

    for (i=0;i<size;i++)
        unsortedArray[i] = input[i].left;
    // fetch left edges

    SortArray (unsortedArray, sortedArray, size);
    // sort left edges ascending
    for (i=0; i<size; i++)
    {
        sweep = sortedArray[i];
        // go out-side-in

        cut = 0;
        for(j=0;j<size;j++)
        {
            if( (input[j].left <= sweep) && (sweep <= input[j].right))
                cut++;
```

```
        }
        // count the intersections with the intervals

        if (cut >= minCut) break;
        // stop when suffiencient many intersections
    }

    if (cut < minCut)
    {
        delete[] sortedArray;
        delete[] unsortedArray;
        return False;
    }
    else
        output.left = sweep;
    // hand back the left edge

    // --- determine the right edge

    for (i=0;i<size;i++)
        unsortedArray[i] = input[i].right;
    // fetch right edges

    SortArray (unsortedArray, sortedArray, size);
    // sort right edges ascending

    for (i=size-1; i>=0; i--)
    {
        sweep = sortedArray[i];
        // go out-side-in

        cut = 0;
        for(j=0;j<size;j++)
        {
            if( (input[j].left <= sweep) && (sweep <= input[j].right))
                cut++;
        }
        // count the intersection with the intervals

        if (cut >= minCut) break;
        // stop when suffiencient many intersections
    }

    if (cut < minCut)
    {
        delete[] sortedArray;
        delete[] unsortedArray;
        return False;
    }
    else
        output.right = sweep;
    // hand back the right edge

    output.ref = (output.left + output.right)/2;
    // take the midpoint, but has no meaning

    delete[] sortedArray;
    delete[] unsortedArray;

    return True;
}


//---------------------------------------------------------------------------
//--- sorting an array of doubles
//---------------------------------------------------------------------------

void SortArray (double* unsortedArray, double* sortedArray, INT32 size)
{
```

```
    Assert (unsortedArray != NULL);
    Assert (sortedArray != NULL);
    Assert (size > 0);

    int i;
    int free;
    int up, low, curr;
    double item;

    free = 0;
    low = up = curr = 0;

    for (i=0;i<size;i++)
    {
        item = unsortedArray[i];

        if (free != 0)  // find the entry
        {
            up = (free>0) ? free-1 : 0;  // prepare for binary search
            low = 0;
            curr = (up+low)/2;

            while ((item != sortedArray[curr]) && (low < up))
            {
                if (item < sortedArray[curr]) up = (curr>0) ? curr-1 : 0;
                else if (item > sortedArray[curr]) low = curr+1;

                curr = (low+up)/2;
            }

            if (item > sortedArray[curr]) curr++;
            // didn't find the entry, need to insert behind the current one
        } // if

        Assert (curr < (SINT32)(size));
        Assert (curr >= 0);

        memmove (sortedArray+curr+1, sortedArray+curr,  // move everything
                 (size-curr-1)*sizeof(double));

        sortedArray[curr] = item;  // enter the timestamp

        free++;  // increment the free index
    } // for
}


//---------------------------------------------------------------------------
//--- intersection of two intervals
//---------------------------------------------------------------------------

EBool Intersect (const SAsymInterval& In1, const SAsymInterval& In2,
    SAsymInterval& Out)
{
    if (In1.left > In2.left)
        Out.left = In1.left;
    else
        Out.left = In2.left;
    // left edge

    if (In1.right > In2.right)
        Out.right = In2.right;
    else
        Out.right = In1.right;
    // right edge

    if (Out.left > Out.right)
        return False;
    // valid interval?
```

```
    Out.ref = (Out.left + Out.right)/2;
    // take the midpoint, but has no meaning

    return True;
}

//---------------------------------------------------------------------------
//--- convert computed state interval to adjusted clock
//---------------------------------------------------------------------------

SExtendedTimestamp IntervalToClock (CNtpTime point, double alpha_minus,
    double alpha_plus)
{
    SExtendedTimestamp clock;         // resulting clock
    CNtpTime a_ntp_plus, a_ntp_minus; // converted accuracies

    clock = point;
    // set the reference point, = is overloaded

    SecondsToNtp (alpha_minus, a_ntp_minus.ms, a_ntp_minus.ts, a_ntp_minus.us);
    SecondsToNtp (alpha_plus, a_ntp_plus.ms, a_ntp_plus.ts, a_ntp_plus.us);
    // map both accuracy intervals into registers

    if ((a_ntp_plus.ms != 0) || (a_ntp_plus.ts >= HighestAccuracyValue))
        a_ntp_plus.ts = Infinity;
    // prepare accuracy overflow

    if ((a_ntp_minus.ms != 0) || (a_ntp_minus.ts >= HighestAccuracyValue))
        a_ntp_minus.ts = Infinity;
    // prepare accuracy overflow

    clock.acc = (a_ntp_plus.ts>>1 & 0x0FFFF) +
                (a_ntp_minus.ts<<15 & 0xFFFF0000);
    // compose the accuracies
    // positive: .ts is (7,-24), need (-8,-23) in lower 2 bytes, hence >>1
    // negative: .ts is (7,-24), need (-8,-23) in upper 2 bytes, hence <<15

    return clock;
}
```

E SimUTC/Sources/Evalsys/evalfunc.cpp

```
//
// evalfunc.cpp
//
// provides functions for the evaluation
//
// created: 01.02.97
//
// last change: 05.02.99
//
// RCS version control information:
//     $Revision: 1.25 $$State: Exp $$Date: 1999/02/05 15:34:47 $
//


#include <sys/socket.h>
#include <sys/types.h>
#include <errno.h>
#include <stream.h>    // must stand *before* iostream, otherwise 'output'...
#include <iostream.h>  // ...is not defined!
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#include "bool.hpp"
#include "constant.hpp"
#include "error.hpp"
#include "event.hpp"
#include "sockstre.hpp"
#include "structs.hpp"
#include "time.hpp"

#include "evalcfg.hpp"   // configuration (all necessary globals)
#include "evalconv.hpp"  // Convert() functions for all events
#include "evalfunc.hpp"  // prototypes of useful functions
#include "evalglob.hpp"  // structure for receiving replies

#ifndef DOS_SIM
  #include <sys/time.h>
#endif

#ifdef PLOT_DATA
  #define PLOT_DIR "gnuplot/"
  // this define is put before the names of the files needed to produce
  // the gnuplot data
  // note: do not forget the slash after the name! if you want the files
  // in the working directory, then you can leave the string empty ("")
#endif


#ifndef NDEBUG
  extern CStreamServer *serverSocket;
  // necessary to remove the server socket before an Assert() fails

  #ifdef Assert
    #undef Assert
  #endif
  #define Assert(expr) \
    if (!(expr)) \
    { \
        (void)delete serverSocket; \
        (void)ReportError (Asserting, __FILE__, __LINE__, #expr); \
    } // if
#endif
```

```
//***************************************************************************
// necessary globals
//***************************************************************************

static double piMin = 1;  // minimum precision of all snapshot rounds
static double piMax = 0;  // maximum precision of all snapshot rounds
static double piSum = 0;  // sum of all precisions (from all snapshot rounds)
static INT32 piRoundNumber = 0;  // number of encountered snapshot rounds
// best, worst and average precision (average = piSum/piRoundNumber)
// note: in fact, the piMin/Max/Sum values are not from all rounds,
// but their computation is started from a user-definable round on
// (the user sets the synchronization round in SEvalsysConfigData, and
// from this value the first valid snapshot round is computed)
// if there are not enough rounds to compute precision values, function
// PrintPrecision() states that it has not collected any data


static INT32 g_Convert (INT32 src);
// if g_byteOrderReversed is True, the reverted value of 'src' is returned;
// if it is False, 'src' is returned unchanged

#ifdef PLOT_DATA
  // used for the output of simulation data
  static ostream outPik[numberOfNodes-1];  // k-th precision
  static ostream outState[numberOfNodes];  // state
  static ostream outAdj[numberOfNodes];  // adjustments
  static ostream outAcc[numberOfNodes];  // accuracy
  static ostream outRate[numberOfNodes];  // rates
  static ostream outGammak[numberOfNodes-1];
  static ostream outDrift[numberOfNodes];  // drifts
  static ostream outMult[numberOfNodes];

  const double TwoPowMinus23 = 1.192092e-07;
  // factor for accuracy

  extern char *ftoa (double f, int width, char *string);  // from ftoa.cpp
  // returns 'f' as a string with at least 'width' characters in %e format;
  // stores the string in 'string', which must be big enough to hold the
  // converted string; if 'string' is NULL, NULL is returned
  // don't forget the link and entry in the Makefile!

  static void SortArray (double* unsortedArray,double* sortedArray,INT32 size);
  // sort an array of doubles

  static double PrecisionKth (double* pi, INT32 upper, INT32 order);
  // computes the kth precision among pi[0],...,pi[upper-1] which
  // are supposed to be sorded ascendingly
  // if order=0 -> maximum distance
  // if order=1 -> 2nd maximum distance
  // etc.
#endif


//***************************************************************************
// function implementations
//***************************************************************************

//---------------------------------------------------------------------------
// message transmission
//---------------------------------------------------------------------------

// send the message (message header and data) over the socket
static void SendData (CStreamInterface *sock, EMessageType msgheader,
                      SBase *data)
{
    Assert (sock != NULL);
    sock->SendMessage (msgheader, data);
}
```

```
//------------------------------------------------------------------------
// parameter/command transmission
//------------------------------------------------------------------------

// send the command over the socket
void ExecuteCommand (CStreamInterface *sock, SCommand *command)
{
    SendData (sock, MsgCommand, command);
}


// send the parameter over the socket
void SetParameter (CStreamInterface *sock, SParameter *param)
{
    SendData (sock, MsgSetParameter, param);
}


// send the parameter over the socket, get the reply, modify param->value
EBool GetParameter (CStreamInterface *sock, SSingleParameter *param)
{
    SendData (sock, MsgGetParameter, param);
    SStructureReply *reply;
    if (ReceiveReply (sock, (SReply**)&reply))
    {
        param->value = ((SSingleParameter*)(reply->data))->value;
        delete reply;
        return True;
    }
    else return False;
}


//------------------------------------------------------------------------
// file output in gnuplot format
//------------------------------------------------------------------------

// prints the line "<x> <y>" on stream 'out'
static void PlotPoint (ostream& out, double x, double y)
{
    //out << x << " " << y << endl;
    // note: this fast version does not work; the plot is not correct

    char buf[40];

    ftoa (x, 14, buf);
    out << buf << " ";

    ftoa (y, 14, buf);
    out << buf << endl;
}

// prints the line "<x> <y> <l> <u>" on stream 'out'
static void PlotInterval (ostream& out, double x, double y, double l, double u)
{
    //out << x << " " << y << " " << l << " " << u << endl;
    // note: this fast version does not work; the plot is not correct

    char buf[40];

    ftoa (x, 14, buf);
    out << buf << " ";

    ftoa (y, 14, buf);
    out << buf << " ";

    ftoa (l, 14, buf);
    out << buf << " ";
```

```
    ftoa (u, 14, buf);
    out << buf << endl;
}


//---------------------------------------------------------------------------
// precision management
//---------------------------------------------------------------------------

static INT32 FirstSnapshotPrecisionRound =
    (INT32)((evalsysConfigData.FirstSyncPrecisionRound*
             simutcConfigData.commonNodeData.RoundPeriod)/
             evalsysConfigData.SnapshotPeriod);
// this is the first snapshot round that is taken into consideration when
// computing the precision
// do not confuse it with the synchronization round: the number of snapshot
// rounds depends on the SnapshotPeriod of the Supervisor triggering the
// snapshots, whereas the number of sync. rounds depends on the RoundPeriod
// of the CSAs

// compute precision per round, store it in global precision data
// piMin, piMax, piSum and piRoundNumber
static void AddPrecision (double accuracy, double simTime)
{
    static double currTime = -1;
    // the time of the current snapshot round; it is set to -1 at the
    // beginning to indicate that the initial values of currPiMax/Min are
    // not valid; from then on, it is always set to the 'simTime' value
    // of the last call

#ifdef PLOT_DATA
    static double TradAcc[numberOfNodes];
    static double TradAccSorted[numberOfNodes];
    static int fill = 0;

    char buf[40];
#else
    static double currAlphaMin;  // minimum accuracy of current round
    static double currAlphaMax;  // maximum accuracy of current round
#endif

    // the following code is parted into two blocks: one if we start
    // a new snapshot round, and one if the accuracy belongs to the
    // same round as that of the previous call

    if (simTime != currTime)  // we start a new snapshot round
    {
        piRoundNumber++;  // increment the round counter

        // we start a new round, so we have to check if the old round
        // was valid; if it was, we have to compute the precision from
        // the minimum and maximum accuracies that we have computed
        // for this round; the precision only affects the global statistics
        // if the snapshot round should be counted

        if ((currTime != -1) &&  // the old round has left valid data
            (piRoundNumber > FirstSnapshotPrecisionRound))  // round is counted
        {
            // if the last round was valid and the round should be counted,
            // we have to compute the precision of this previous round and
            // this value is taken over into the global variables
            // piMin/Max/Sum

            double piLastRound;   // precision of the previous round

#ifdef PLOT_DATA
            SortArray (TradAcc, TradAccSorted, fill);
```

```
            piLastRound = PrecisionKth (TradAccSorted, fill, 0);

            // output simulation data
            for (int i=0; i<numberOfNodes-1; i++)
            {
                PlotPoint (outPik[i], currTime,
                    PrecisionKth (TradAccSorted, fill, i));
                // plot the time (from the former round) and the i-th
                // precision
            } // for
#else
            piLastRound = currAlphaMax - currAlphaMin;
            // compute the precision of the last round
#endif

            if (piMax < piLastRound) piMax = piLastRound;
            if (piMin > piLastRound) piMin = piLastRound;
            // adapt the minimum and maximum precision if necessary

            piSum += piLastRound;
            // add the precision to the sum value
        } // if

        // now that the old round has been handled, consider the new data
        // we have got
        // remember that we are in a new round; so the 'accuracy' we
        // got is the best and worst accuracy we currently have and
        // we have to store it as minimum and as maximum

#ifdef PLOT_DATA
        fill = 0;
        TradAcc[fill] = accuracy;
        fill++;
        // first entry for the new snapshot
#else
        currAlphaMax = accuracy;  // store the minimum accuracy
        currAlphaMin = accuracy;  // store the maximum accuracy
#endif

        currTime = simTime;  // store the time of the new snapshot round
    }
    else  // the data belongs to the current snapshot round
    {
#ifdef PLOT_DATA
        Assert( fill < numberOfNodes );
        TradAcc[fill] = accuracy;
        fill++;
#else
        // if the accuracy belongs to the same round as that of the
        // previous call, we just have to update the minimum/maximum
        // accuracy values if necessary

        if (currAlphaMax < accuracy) currAlphaMax = accuracy;
        // if the 'accuracy' is greater than the current maximum -> store it

        if (currAlphaMin > accuracy) currAlphaMin = accuracy;
        // if the 'accuracy' is less than the current minimum -> store it
#endif
    } // else
}


//-----------------------------------------------------------------------
// consonance management
//-----------------------------------------------------------------------

#ifdef PLOT_DATA
// compute consonance per round, store it in global consonance data
static void AddConsonance (double rate, double simTime)
```

```
{
    static double currTime = -1;
    // the time of the current snapshot round; it is set to -1 at the
    // beginning to indicate that the initial values of currPiMax/Min are
    // not valid; from then on, it is always set to the 'simTime' value
    // of the last call

    static double CurrentRate[numberOfNodes];
    static double CurrentRateSorted[numberOfNodes];
    static int fill = 0;

    char buf[40];

    if (simTime != currTime)  // new snapshot round
    {
        if (currTime != -1)   // compute consonance of previous round
        {
            double consLastRound;

            SortArray (CurrentRate, CurrentRateSorted, fill);

            consLastRound = PrecisionKth (CurrentRateSorted, fill, 0);

            for (int i=0; i<numberOfNodes-1; i++)
            {
                PlotPoint (outGammak[i], currTime,
                    PrecisionKth (CurrentRateSorted, fill, i)*1e6);
                // plot the time (from the former round) and the i-th
                // consonance
            } // for
        } // if

        fill = 0;
        CurrentRate[fill] = rate;
        fill++;
        // first entry for the new snapshot

        currTime = simTime;
        // store the time of the new snapshot round
    }
    else  // the data belongs to the current snapshot round
    {
        Assert( fill < numberOfNodes );
        CurrentRate[fill] = rate;
        fill++;
    }
}
#endif


//------------------------------------------------------------------------
// event printing
//------------------------------------------------------------------------

// print events from the network
// note: you must convert the event, even if its data is not needed;
// the calling function depends on it
static void PrintNetworkEvent (SEvent *event, _IO_ostream_withassign& out)
{
    SNetEvent *netev = (SNetEvent *)event;  // basic network event
    SNetMessageEvent *msgev = (SNetMessageEvent *)event;
    SNetCrashEvent *netcrash = (SNetCrashEvent *)event;

    out << "network (" << g_Convert (netev->networkID) << "): ";
    // every network event output starts with the network ID

    switch (g_Convert (event->type))
    {
        case CrashStatusChange:
```

```
            Convert (netcrash, g_byteOrderReversed);
            if (netcrash->crashed) out << "crashed";
            else out << "recovered";
            break;
        case DeliverMessage:
            Convert (msgev, g_byteOrderReversed);
            out << "delivered message (" << msgev->sender;
            out << ", " << msgev->area << ", " << msgev->receiver;
            out << ", " << msgev->msgID << ")";
            break;
        case EarlyMessage:
            Convert (msgev, g_byteOrderReversed);
            out << "sent early message (" << msgev->sender;
            out << ", " << msgev->area << ", " << msgev->receiver;
            out << ", " << msgev->msgID << ")";
            break;
        case LateMessage:
            Convert (msgev, g_byteOrderReversed);
            out << "sent late message (" << msgev->sender;
            out << ", " << msgev->area << ", " << msgev->receiver;
            out << ", " << msgev->msgID << ")";
            break;
        case MessageLost:
            Convert (msgev, g_byteOrderReversed);
            out << "discarded message (" << msgev->sender;
            out << ", " << msgev->area << ", " << msgev->receiver;
            out << ", " << msgev->msgID << ")";
            break;
        case SendMessage:
            Convert (msgev, g_byteOrderReversed);
            out << "transmitted message (" << msgev->sender;
            out << ", " << msgev->area << ", " << msgev->receiver;
            out << ", " << msgev->msgID << ")";
            break;

        default:
            Convert (netev, g_byteOrderReversed);
            out << "unknown event: " << event->type;
            break;
    } // switch
}


// converts a module number into a string constant
// (same function as in error.cpp)
static const char *ModuleToString (EModule module)
{
    switch (module)
    {
        case Controller: return "Controller"; break;
        case Network: return "Network"; break;
        case Supervisor: return "Supervisor"; break;
        case ClockSync: return "ClockSync"; break;
        case Clock: return "Clock"; break;
        case Lance: return "Lance"; break;

        default: return "Unknown"; break;
    } // switch
}

// print the data of one message
// format: "message <msgID> (<sender>,<area> => <receiver>) "
static void PrintMsg (SINT32 msgID, SINT32 sender, int area, SINT32 receiver,
                      _IO_ostream_withassign& out)
{
    out << "message " << msgID << " (" << sender << "," << area;
    out << " => " << receiver << ") ";
}
```

```
// print events from the CSA
// note: you must convert the event, even if its data is not needed;
// the calling function depends on it
static void PrintCSAEvent (SEvent *event, _IO_ostream_withassign& out)
{
    SApplEvent *appev = (SApplEvent *)event;
    SApplClockEvent *ev = (SApplClockEvent *)event;
    SApplPreloadEvent *pre = (SApplPreloadEvent *)event;
    SApplGpsEvent *gpsev = (SApplGpsEvent *)event;
    SApplCrashEvent *crashev = (SApplCrashEvent *)event;
    SApplSendMsgEvent *appsmsg = (SApplSendMsgEvent *)event;
    SApplRecvMsgEvent *apprmsg = (SApplRecvMsgEvent *)event;
    SApplDelayEvent *appdelay = (SApplDelayEvent *)event;
    SApplDelayEstimationEvent *appest = (SApplDelayEstimationEvent *)event;
    SApplNetEvent *netev = (SApplNetEvent *)event;
    SApplTimeEvent *intev = (SApplTimeEvent *)event;
    SApplRateEvent *rateev = (SApplRateEvent *)event;

#ifdef PLOT_DATA
    int nodeIndex;
    double tradacc;
    double edge_lower, edge_upper;
    double rate;
    static double state_old[numberOfNodes];
    static double time_old[numberOfNodes];
    static EBool inhibit[numberOfNodes];
    static EBool firstCall = True;

    if (firstCall)  // initialize the inhibit[] array
    {
        for (int i=0; i<numberOfNodes; i++) inhibit[i] = True;
        firstCall = False;
    } // if
#endif

    out << "node (" << g_Convert (appev->nodeID);
    out << "," << g_Convert (appev->area) << "): ";
    // every CSA event output starts with the node ID


    // check if we should update the round counter
    if ((g_Convert(appev->type) == evalsysConfigData.RoundEvent) &&
        (g_Convert(appev->nodeID) == evalsysConfigData.RoundNodeID) &&
        (g_Convert(appev->area) == evalsysConfigData.RoundArea))
    {
        round++; // update round counter
    } // if


    switch (g_Convert(event->type))
    {
        case ChangeAssociation:  // associated network was changed
            Convert (netev, g_byteOrderReversed);
            out << "associated network set to " << netev->networkID;
            break;
        case ClockPreload:  // this is a clock preload event
            Convert (pre, g_byteOrderReversed);
            out << "clock preloaded from ";
            out << pre->loadAt << " to ";
            out << NtpToSeconds (pre->newTime);
            if ((ev->nodeID == evalsysConfigData.RoundNodeID) &&
                (ev->area == evalsysConfigData.RoundArea))
                round++; // update round counter
            break;
        case ClockReadFault:  // clock returned wrong value on read access
            Convert (appev, g_byteOrderReversed);
            out << "clock return arbitrary value on read";
            break;
        case ClockUpdate:  // this is a clock event
```

```
            Convert (ev, g_byteOrderReversed);
            out << "clock set from ";
            out << NtpToSeconds (ev->clockTime);
            out << " to ";
            out << NtpToSeconds (ev->newTime);

#ifdef PLOT_DATA
            // --- output simulation data

            nodeIndex = ev->nodeID-1;
            Assert (nodeIndex >= 0);
            // index is nodeID-1, so it starts from 0

            PlotPoint (outAdj[nodeIndex], event->time,
                NtpToSeconds (ev->newTime) - NtpToSeconds (ev->clockTime));
            // print the pair <time, adjustment>

            inhibit[nodeIndex] = True;
            // skip the next snapshot
#endif

            break;
        case ClockWriteFault:  // clock was set to wrong value on write access
            Convert (appev, g_byteOrderReversed);
            out << "clock is set to wrong value on write";
            break;
        case ConvergenceFailed:  // convergence function failed
            Convert (appev, g_byteOrderReversed);
            out << "convergence function failed";
            break;
        case CrashStatusChange:  // crash status has changed
            Convert (crashev, g_byteOrderReversed);
            if (crashev->crashed) out << "crashed";
            else out << "recovered";
            break;
        case DelayEstimation:  // new estimated transmission delay
            Convert (appest, g_byteOrderReversed);
            out << "estimated delay from " << appest->sender;
            out << " on network " << appest->networkID << ": ";
            out << "(" << appest->delay.delta << "; ";
            out << appest->delay.delta - appest->delay.epsilonMinus << ", ";
            out << appest->delay.delta + appest->delay.epsilonPlus << ")";
            break;
        case DiscardRecvMessage:  // received message was discarded
            Convert (apprmsg, g_byteOrderReversed);
            PrintMsg (apprmsg->msgID, apprmsg->sender, apprmsg->senderArea,
                      apprmsg->nodeID, out);
            out << " was discarded by receiver";
            break;
        case DiscardSendMessage:  // sent message was discarded
            Convert (appsmsg, g_byteOrderReversed);
            PrintMsg (appsmsg->msgID, appsmsg->nodeID, appsmsg->area,
                      appsmsg->receiver, out);
            out << " was discarded by sender";
            break;
        case GpsEvent:  // this is a GPS update event
            Convert (gpsev, g_byteOrderReversed);
            out << "clock differs from GPS time by ";
            out << gpsev->diff << " seconds";
            break;
        case IgnoreSetting:  // clock has ignored register access
            Convert (appev, g_byteOrderReversed);
            out << "clock has ignored register access";
            break;
        case IntervalEvent:  // time interval from convergence function
            Convert (intev, g_byteOrderReversed);
            if (intev->input) out << "CV got input interval (";
            else out << "CV computed interval (";
            out << NtpToSeconds (intev->interval.low) << ", ";
```

```
            out << NtpToSeconds (intev->interval.high) << "; ";
            out << NtpToSeconds (intev->interval.refpoint) << ")";
            break;
        case RateEvent:  // rate interval from rate algorithm
            Convert (rateev, g_byteOrderReversed);
            if (rateev->input) out << "rate alg. got input rate (";
            else out << "rate alg. computed rate (";
            out << rateev->rate.thetaMinus << ", ";
            out << rateev->rate.thetaPlus << "; ";
            out << rateev->refPoint << "; ";
            out << rateev->rate.u << ")";

#ifdef PLOT_DATA
            // --- output simulation data

            edge_lower = 1.0 - rateev->rate.thetaMinus;
            edge_upper = 1.0 + rateev->rate.thetaPlus;
            // edges of the local rate interval

            nodeIndex = rateev->nodeID-1;
            Assert (nodeIndex >= 0);
            // index is nodeID-1, so it starts from 0

            if (!rateev->input)   // only use the computed rate
            {
                PlotInterval (outDrift[nodeIndex], event->time + nodeIndex*1.0,
                    1.0, edge_lower, edge_upper);
                // put some space between the nodes, otherwise their
                // data is plotted at the same x-position

                PlotPoint (outMult[nodeIndex], event->time,
                    (rateev->refPoint - 1) * 1e6);
            } // if
#endif

            break;
        case ReceivedMessage:  // message was received
            Convert (apprmsg, g_byteOrderReversed);
            PrintMsg (apprmsg->msgID, apprmsg->sender, apprmsg->senderArea,
                      apprmsg->nodeID, out);
            out << " was received";
            break;
        case ScrambleRecvMessage:  // incoming message was scrambled
            Convert (apprmsg, g_byteOrderReversed);
            PrintMsg (apprmsg->msgID, apprmsg->sender, apprmsg->senderArea,
                      apprmsg->nodeID, out);
            out << " was scrambled by receiver";
            break;
        case ScrambleSendMessage:  // outgoing message was scrambled
            Convert (appsmsg, g_byteOrderReversed);
            PrintMsg (appsmsg->msgID, appsmsg->nodeID, appsmsg->area,
                      appsmsg->receiver, out);
            out << " was scrambled by sender";
            break;
        case DuplicateRecvMessage:  // incoming message was duplicated
            Convert (apprmsg, g_byteOrderReversed);
            PrintMsg (apprmsg->msgID, apprmsg->sender, apprmsg->senderArea,
                      apprmsg->nodeID, out);
            out << " was duplicated by receiver";
            break;
        case DuplicateSendMessage:  // outgoing message was duplicated
            Convert (appsmsg, g_byteOrderReversed);
            PrintMsg (appsmsg->msgID, appsmsg->nodeID, appsmsg->area,
                      appsmsg->receiver, out);
            out << " is a duplication at sender";
            // note: the msg ID is that of the duplicated message, not
            // that of the original message (however, most of the time,
            // the msg ID of the original msg will be msgID-1)
            break;
```

```
        case SendMessage:  // message was sent
            Convert (appsmsg, g_byteOrderReversed);
            PrintMsg (appsmsg->msgID, appsmsg->nodeID, appsmsg->area,
                      appsmsg->receiver, out);
            out << " was sent";
            break;
        case Snapshot:  // system snapshot was taken
            Convert (ev, g_byteOrderReversed);
            out << "clock made snapshot ";
            out << NtpToSeconds (ev->newTime);
            out << " (acc = [" << ((ev->newTime.acc) & 0x0FFFF) << ", ";
            out << ((ev->newTime.acc>>16) & 0x0FFFF) << "])";
            AddPrecision (ev->time - NtpToSeconds (ev->newTime), ev->time);

#ifdef PLOT_DATA
            // --- output simulation data

            nodeIndex = ev->nodeID-1;
            Assert (nodeIndex >= 0);
            // index is nodeID-1, so it starts from 0

            tradacc = NtpToSeconds (ev->newTime) - event->time;
            // current tradition accuracy

            edge_lower = tradacc -
                (double)((ev->newTime.acc >> 16) & 0x0FFFF) * TwoPowMinus23;
            edge_upper = tradacc +
                (double)((ev->newTime.acc) & 0x0FFFF) * TwoPowMinus23;
            // lower/upper accuracies

            if (!inhibit[nodeIndex])  // rate can be computed
                rate = (NtpToSeconds (ev->newTime) - state_old[nodeIndex]) /
                    (event->time - time_old[nodeIndex]);
            else
                rate = 1.0;
            // current rate

            state_old[nodeIndex] = NtpToSeconds (ev->newTime);
            time_old[nodeIndex]  = event->time;
            // remember for the next snapshot

            PlotPoint (outState[nodeIndex], event->time, tradacc);

            PlotInterval (outAcc[nodeIndex], event->time + nodeIndex*0.4,
                tradacc, edge_lower, edge_upper);


            if (!inhibit[nodeIndex])  // print the rate
            {
                PlotPoint (outRate[nodeIndex], event->time,
                    (rate-1) * 1e6);
                AddConsonance (rate, ev->time);
            }
            else inhibit[nodeIndex] = False;
            // from the second call on, we can print the rate
#endif

            break;
        case TimerEvent:  // a clock timer has produced interrupt
            Convert (appev, g_byteOrderReversed);
            out << "duty timer interrupt for module ";
            out << ModuleToString (appev->module);
            break;
        case TransmissionDelay:  // measured transmission delay
            Convert (appdelay, g_byteOrderReversed);
            PrintMsg (appdelay->msgID, appdelay->sender, appdelay->senderArea,
                      appdelay->nodeID, out);
            out << " was transmitted in " << NtpToSeconds(appdelay->delaytime);
            out << " seconds";
```

```
            break;
        case UserEvent:  // contains user-specific data
            Convert (appev, g_byteOrderReversed);
            out << "user defined event occured";

        default:
            Convert (appev, g_byteOrderReversed);
            out << "unknown event: " << event->type;
            break;
    } // switch
}


// print the event data to cout
static void PrintEvent (SEvent *event)
{
    SApplEvent *appev;  // for events from a CSA

    _IO_ostream_withassign out;  // output stream

#ifdef NO_PRINTEVENT  // redirect event output to /dev/null
    filebuf fbuf;
    if (fbuf.open ("/dev/null", "w") == 0)
    {
        cerr << "cannot open /dev/null" << endl;
        out = cout;  // use cout instead
        out.setf (ios::fixed, ios::floatfield);
        // always print 6 (default) digits
    }
    else
    {
        ostream tmpout (&fbuf);  // output stream
        out = tmpout;
    }
#else
    out = cout;
    out.setf (ios::fixed, ios::floatfield);
    // always print 6 (default) digits
#endif

    switch (g_Convert(event->module))
    {
        case Network:
            PrintNetworkEvent (event, out);
            // the call also converts the event if necessary
            break;
        case Lance:
        case ClockSync:
        case Clock:
            PrintCSAEvent (event, out);
            // the call also converts the event if necessary
            break;

        default:
            Convert (event, g_byteOrderReversed);  // convert base event
            out << "module not supported: " << event->module;
   } // switch

   out << " at simulation time " << event->time;

   switch (g_Convert(event->module))
   {
        case Lance:      // convert it to an application event, extract
        case ClockSync:  // the clock time
        case Clock:
            appev = (SApplEvent *)event;
            out << " (";
            out << NtpToSeconds (appev->clockTime);
            out << ")";
```

```
            break;
    } // switch

    out << endl;
}


//-----------------------------------------------------------------------
// message reception
//-----------------------------------------------------------------------

// returns True if the message with the appropriate header was received,
// otherwise False; if the message was received sucessfully, it is returned
// in 'buf'
EBool ReceiveMessage (CStreamInterface *sock, EMessageType header,
                      SBase **buf)
{
    Assert (sock != NULL);
    Assert (buf != NULL);

    EMessageType type;
    int failureCount = 0;  // number of different messages encountered

    do {
        // wait for the reply (block for BlockTime seconds)
        if (!sock->ReceiveMessageHeader (&type, evalsysConfigData.BlockTime))
        {
            // that can only be EOF (socket killed???)
            // (when using timeouts, it can also be a timeout)
            Report (evalsys: encountered timeout or EOF reading header);
            return False;
        }

        if (type != header)
        {
            if (failureCount == 0)
            {
                cerr << "evalsys: encountering unexpected message(s)." << endl;
            }
            failureCount++;

            if (sock->ReceiveMessage (buf))
            // read a message if it's there, process it
            {
                switch (type)
                {
                    case MsgEvent:
                        PrintEvent ((SEvent *)(*buf));
                        break;
                    case MsgReply:
                        cerr << "evalsys: was waiting for header " << header;
                        cerr << ", have received a reply" << endl;
                        break;

                    default:
                        cerr << "evalsys: was waiting for header " << header;
                        cerr << ", have received " << type << endl;
                        break;
                } // switch

                sock->ReturnMessageBuffer (*buf);  // return the buffer
            } // if
        } // if
        else if (!sock->ReceiveMessage (buf))
        {
            return False;
        }
    } while (type != header);
```

```
    if (failureCount > 0)
    {
        cerr << "evalsys: encountered " << failureCount << " different";
        cerr << " messages before reading the expected message" << endl;
    }
    return True;
}


//-----------------------------------------------------------------------
// special message reception
//-----------------------------------------------------------------------

// receives an event and prints it to cout; returns True if the event
// was received sucessfully, else False
EBool ReceiveEvent (CStreamInterface *sock)
{
    SBase *buf;  // msg buffer

    if (!ReceiveMessage (sock, MsgEvent, &buf))
    {
        Report (evalsys: msg was no event);
        return False;
    }

    PrintEvent ((SEvent *)buf);
    // assume that the buffer contains an event

    sock->ReturnMessageBuffer (buf);
    // return the buffer

    return True;
}


// receives a reply, allocates the appropriate reply structure and returns
// it in 'reply'; returns True if the reply was received successfully,
// else False
EBool ReceiveReply (CStreamInterface *sock, SReply **reply)
{
    Assert (sock != NULL);
    Assert (reply != NULL);

    SBase *buf;  // msg buffer

    if (!ReceiveMessage (sock, MsgReply, &buf))
    {
        Report (evalsys: msg was no reply);
        return False;
    }

    if (buf->size < sizeof(SReply))
    {
        Report (evalsys: reply was not complete);
        sock->ReturnMessageBuffer (buf);
        return False;
    }

    char *param;  // for structure replies
    INT32 size;  // size of the param structure

    switch (g_Convert(((SReply*)buf)->type))
    {
        case ReplyBoolean:
            *reply = new SBooleanReply;
            memcpy (*reply, buf, buf->size);  // copy type and data
            Convert ((SBooleanReply *)*reply, g_byteOrderReversed);
            break;
        case ReplyDouble:
```

```
            *reply = new SDoubleReply;
            memcpy (*reply, buf, buf->size);  // copy type and data
            Convert ((SDoubleReply *)*reply, g_byteOrderReversed);
            break;
        case ReplyStructure:
            size = buf->size - sizeof(SReply);
            // size of the param structure
            param = new char[size];
            memcpy (param, (char*)buf+sizeof(SReply), size);
            // copy the parameter structure
            *reply = new SStructureReply (param, size);
            Convert ((SStructureReply *)*reply, g_byteOrderReversed);
            break;

        default:  // unknown type
            Report (evalsys: reply type was unknown);
            sock->ReturnMessageBuffer (buf);
            return False;
    } // switch

    sock->ReturnMessageBuffer (buf);  // not needed anymore
    return True;
}


// waits for a reply, prints an error message containing 'command' to cerr
// if (a) there was no reply, (b) it was no boolean reply, or (c) the reply
// contained the boolean value False
void CheckBooleanReply (CStreamInterface *sock, const char *command)
{
    SBooleanReply *boolReply;
    if (!ReceiveReply (sock, (SReply **)&boolReply))
    {
        cerr << "error receiving a reply" << endl;
        return;
    }
    if (boolReply->type != ReplyBoolean) Report (evalsys: wrong reply!);
    if (boolReply->result != True)
    {
        cerr << command << ": have received the reply ";
        cerr << boolReply->result << endl;
    }
    delete boolReply;
}


//---------------------------------------------------------------------------
// socket management
//---------------------------------------------------------------------------

#ifndef DOS_SIM
// select() is only known in UNIX, not in DOS

// waits for the supervisor sockets to become readable (using select);
// returns the index of the first socket that is readable (if more than one
// socket is readable, the smallest index is returned)
int SelectSocket ()
{
    fd_set readSet;  // read mask for select()
    timeval tv;  // for the timeout of select()
    int err;  // return value of select()
    static INT32 maxsid=0;  // the highest supervisor socket ID

    FD_ZERO (&readSet);  // initialize the set

    for (int i=0; i<numberOfNodes+1; i++)
    {
        FD_SET (supsid[i], &readSet);
        // set the read mask for the socket IDs
```

```
        if (supsid[i] > maxsid) maxsid = supsid[i];
        // find out the highest socket ID
    }

    tv.tv_sec  = (long)evalsysConfigData.SelectTimeout;  // extract the seconds
    tv.tv_usec = (long)(evalsysConfigData.SelectTimeout*1e6) % (long)1e6;
    // extract the microseconds

    // now wait for the socket to become readable
    err = select (maxsid+1, &readSet, 0, 0, &tv);

    if (err == -1)
    {
        Report (error using select);
        return NoAccount;
    }
    if (err == 0) return NoAccount;  // timeout

    // now check the read mask for the first socket that is set
    for (int i=0; i<numberOfNodes+1; i++)
    {
        if (FD_ISSET (supsid[i], &readSet)) return i;
    }

    return NoAccount;  // should never happen
}

#else
int SelectSocket ()  // just some dummy function
{
    return NoAccount;
}
#endif


// checks if the socket with the given address already exists and returns
// True if the check succeeds, False if the socket does not exist
// the function performs its check by creating a socket with this address
// and type and by checking if this creation (call to s_create()) fails;
// if it fails, the function returns True, otherwise it deletes the socket
// again and returns False
EBool SocketExists (const ACSocketAddress *addr, ESocketType type)
{
    Assert (addr != NULL);

    INT32 sid;

    if (s_create (addr, type, &sid) != SOkay) return True;
    // note: this also returns True if the socket could not be created
    // due to some other reason!

    // if we come here, then the socket was created successfully
    s_delete (sid, addr);  // close the socket again
    return False;
}


//---------------------------------------------------------------------------
// 'normalize' a double time value so that it is > 1
//---------------------------------------------------------------------------

// converts 'input' (in seconds) to a double >= 1, stores the unit
// needed in 'unit'; unit should be at least 8 characters wide
static void Normalize (double input, double& output, char *unit)
{
    int i;
    short sign = 0;  // sign of 'input'; if input < 0 -> sign = 1
    char *unitname[5] = { "s", "ms", "us", "ns", "ps" };
```

```
    // the unit of the output value

    if (input == 0)  // just copy the zero
    {
        output = 0;
        strcpy (unit, unitname[0]);
        return;
    }

    if (input < 0)  // make input positive (makes computation easier)
    {
        input = -input;
        sign = 1;
    }

    for (i=0; (i < 4) && (input < 1); i++) input = input * 1000;
    // now 'input' is >= 1, and i is the power (base 10)

    if (sign) output = -input;  // copy the result to 'output'
    else output = input;

    // check the power, get the appropriate unit name
    if ((i >= 0) && (i < 5)) strcpy (unit, unitname[i]);
    else strcpy (unit, "unknown");
}


//---------------------------------------------------------------------------
// printing the configuration
//---------------------------------------------------------------------------

// prints the value to 'o' in the form "<param> = <norm> <unit>", where
// 'norm' and 'unit' are obtained from Normalize()
static void PrintNormal (ostream& o, const char* param, double value)
{
    double norm;   // "normalized" number for output
    char unit[8];  // unit name

    Normalize (value, norm, unit);
    o << param << " = " << norm << " " << unit << endl;
}

// prints the string "probability of <prob> = <value>" to 'o'
static void PrintProbability (ostream& o, const char *prob, double value)
{
    o << "probability of " << prob << " = " << value << endl;
}

// prints the configuration data
void PrintConfiguration ()
{
    double norm;   // "normalized" number for output
    char unit[8];  // unit name

    cout << endl << "configuration of the simulation:" << endl;

    if (numberOfNetworks > 0)  // pure simulation configured a network
    {
        cout << endl << "  Networks: " << numberOfNetworks << endl;
        cout << "    All nodes are connected to the first network." << endl;
        cout << "    "; PrintProbability (cout, "network crashes",
                        simutcConfigData.networkData.ProbNetCrash);
        cout << "    "; PrintProbability (cout,"network recovery",
                        simutcConfigData.networkData.ProbNetRecover);
        cout << "    "; PrintProbability (cout, "early messages",
                        simutcConfigData.networkData.ProbNetEarly);
        cout << "    "; PrintProbability (cout, "late messages",
                        simutcConfigData.networkData.ProbNetLate);
        cout << "    "; PrintNormal (cout, "lambda",
```

```
                        simutcConfigData.networkData.Lambda);
      cout << "    "; PrintNormal (cout, "dmin",
                        simutcConfigData.networkData.Dmin);
      cout << "    "; PrintNormal (cout, "dmax",
                        simutcConfigData.networkData.Dmax);
      cout << "    for all nodes i,j with i!=j:" << endl;
      cout << "       "; PrintNormal (cout, "delta",
                           simutcConfigData.networkData.Delta);
      cout << "       "; PrintNormal (cout, "epsMinus",
                           simutcConfigData.networkData.EpsilonMinus);
      cout << "       "; PrintNormal (cout, "epsPlus",
                           simutcConfigData.networkData.EpsilonPlus);
  } // if

  cout << endl << "  Nodes: " << numberOfNodes << endl;
  cout << "    "; PrintNormal (cout, "round period",
                     simutcConfigData.commonNodeData.RoundPeriod);

  cout << "    delay measurement every ";
  cout << simutcConfigData.commonNodeData.DelayCycles << " cycles" << endl;

  cout << "    rate synchronization every ";
  cout << simutcConfigData.commonNodeData.RateCycles << " cycles" << endl;

  cout << "    synchronization algorithm is ";
  switch (simutcConfigData.commonNodeData.SynchronizationAlgorithm)
  {
      case FreeRun: cout << "free-run";
          break;
      case Midpoint: cout << "FTA (using midpoint)";
          break;
      case FaultTolerantAverage: cout << "FTA (using reference point)";
          break;
      case Fetzer: cout << "FTA (reference point) with Fetzer for P-Nodes";
          break;
      case OP: cout << "OP (Optimal Precision)";
          break;
      case Marzullo: cout << "Marzullo";
          break;
      case Rate: cout << "Rate Synchronization";
          break;
      case FaultTolerantIntersection: cout << "FTI (Fault Tolerant"
                                              " Intersection)";
          break;
      default:
          cout << simutcConfigData.commonNodeData.SynchronizationAlgorithm;
          cout << " (unknown)";
  } // switch
  cout << endl;

  cout << endl;
  cout << "    node specific data:" << endl;
  for (int i=0; i<numberOfNodes; i++)
  {
      // if the node data is different or if there is only one node,
      // we print the text "node 1"; if the node data is the same for
      // all nodes and there are at least two, we print "nodes 1-n"
      // where 'n' is the number of nodes
      if (simutcConfigData.nodesDiffer || (numberOfNodes==1))
          cout << "    node " << i+1 << ": " << endl;
      else cout << "    nodes 1-" << numberOfNodes << ": " << endl;

      // now print the node data
      cout << "      "; PrintNormal (cout, "clock drift",
                           simutcConfigData.nodeData[i].ClockDrift);

      cout << "      node type = ";
      if (simutcConfigData.nodeData[i].ConnectedGpus) cout << "P-Node";
      else cout << "S-Node";
```

```
        cout << endl;

        cout << "      "; PrintNormal (cout, "delay compensation",
                             simutcConfigData.nodeData[i].DelayCompensation);
        cout << "      "; PrintNormal (cout, "computation delay compensation",
                             simutcConfigData.nodeData[i].CompDelayCompensation);
        cout << "      "; PrintProbability (cout, "node crashes",
                             simutcConfigData.nodeData[i].ProbLanceCrash);
        cout << "      "; PrintProbability (cout, "node recovery",
                             simutcConfigData.nodeData[i].ProbLanceRecover);
        cout << "      ";PrintNormal(cout, "maximum delay of outgoing messages",
                             simutcConfigData.nodeData[i].DmaxSendLance);
        cout << "      ";PrintNormal(cout, "maximum delay of incoming messages",
                             simutcConfigData.nodeData[i].DmaxRecvLance);
        cout << "      "; PrintProbability (cout, "scrambling send messages",
                             simutcConfigData.nodeData[i].ProbLanceSendScramble);
        cout << "      "; PrintProbability (cout, "scrambling recv messages",
                             simutcConfigData.nodeData[i].ProbLanceRecvScramble);
        cout << "      "; PrintProbability (cout, "duplicating send messages",
                             simutcConfigData.nodeData[i].ProbLanceSendDuplicate);
        cout << "      "; PrintProbability (cout, "duplicating recv messages",
                             simutcConfigData.nodeData[i].ProbLanceRecvDuplicate);


        if (!simutcConfigData.nodesDiffer) break;
        // jump out of the loop if the other nodes have the same data
    } // for

    cout << endl;
}


//-------------------------------------------------------------------------
// printing the measured precision
//-------------------------------------------------------------------------

// prints the precision data collected during evaluation
void PrintPrecision ()
{
    double norm;   // "normalized" number for output
    char unit[8];  // unit name

    cout << endl << "precision data";

    if (piRoundNumber > FirstSnapshotPrecisionRound)
    // the round number always includes the current (possibly unfinished)
    // round; this round is not included in the summary, so there must be
    // at least FirstSnapshotPrecisionRound+1 rounds for valid precision
    // data
    {
        double average;  // the average value

        average = piSum/(piRoundNumber-FirstSnapshotPrecisionRound);

        cout << " (from round ";
        cout << evalsysConfigData.FirstSyncPrecisionRound << " on):" << endl;
        cout << "    snapshots: " << piRoundNumber << endl;

        Normalize (piMin, norm, unit);
        cout << "    minimum: " << norm << " " << unit << endl;
        Normalize (piMax, norm, unit);
        cout << "    maximum: " << norm << " " << unit << endl;
        Normalize (average, norm, unit);
        cout << "    average: " << norm << " " << unit << endl;
    } // if
    else cout << ": none collected" << endl;
}
```

```
//---------------------------------------------------------------------------
// conversion function
//---------------------------------------------------------------------------

// if g_byteOrderReversed is True, the reverted value of 'src' is returned;
// if it is False, 'src' is returned unchanged
static INT32 g_Convert (INT32 src)
{
    if (!g_byteOrderReversed) return src;  // nothing to convert

    INT32 tmp;
    tmp  = (src << 24) & 0xFF000000;
    tmp += (src <<  8) & 0x00FF0000;
    tmp += (src >>  8) & 0x0000FF00;
    tmp += (src >> 24) & 0x000000FF;
    return tmp;
}


#ifdef PLOT_DATA

//---------------------------------------------------------------------------
//--- open files
//---------------------------------------------------------------------------

// used for the output of simulation data
static filebuf Frun;  // LaTeX wrapper file
static ostream outRun;  // stream for Frun
static filebuf Fpik[numberOfNodes-1];  // k-th precision
static filebuf Fstate[numberOfNodes];  // state
static filebuf Fadj[numberOfNodes];  // adjustments
static filebuf Facc[numberOfNodes];  // accuracy
static filebuf Frate[numberOfNodes];  // rates
static filebuf Fgammak[numberOfNodes-1];  // k-th consonance
static filebuf Fdrift[numberOfNodes];  // drifts
static filebuf Fmult[numberOfNodes];


// generates the filename from 'name' followed by 'number'+1, with the
// extension ".dat" in the subdirectory PLOT_DIR, opens the filebuffer
// file[number], associates the buffer with the ostream stream[number]
static void OpenFile (filebuf *file, ostream* stream, int number,
                      const char *name)
{
    static char filename[256];  // for generating the file names
    static char buf[20];  // buffer for itoa()

    strcpy (filename, PLOT_DIR);
    strcat (filename, name);
    strcat (filename, itoa (number+1, buf, 10));
    strcat (filename, ".dat");
    // filenames are generated from the name and the number

    Assert (file[number].open (filename, output) != 0);
    // open the file

    stream[number].rdbuf (&file[number]);
    // associate the filebuffer with its stream
}

// same as OpenFile(), but the filename is generated from 'name' followed
// by 'number' (i.e., starting from 0, as required by k-th order plots)
static void OpenOrderFile (filebuf *file, ostream* stream, int number,
                           const char *name)
{
    static char filename[256];  // for generating the file names
    static char buf[20];  // buffer for itoa()

    strcpy (filename, PLOT_DIR);
```

```
    strcat (filename, name);
    strcat (filename, itoa (number, buf, 10));
    strcat (filename, ".dat");
    // filenames are generated from the name and the number

    Assert (file[number].open (filename, output) != 0);
    // open the file

    stream[number].rdbuf (&file[number]);
    // associate the filebuffer with its stream
}

static void GenerateTexFile();
// prints the TeX-Output into run.tex

// opens all file buffers
void OpenPlotFiles()
{
    int i;
    char filename[256];

    strcpy (filename, PLOT_DIR);
    strcat (filename, "run.tex");

    // open and generate the TeX-File
    Assert (Frun.open (filename, output) != 0);
    GenerateTexFile();

    // for a simulation run output precision by each snapshot
    for (i=0; i<numberOfNodes-1; i++)
    {
        OpenOrderFile (Fpik, outPik, i, "pik");  // plots a k-th order
    } // for

    // for a simulation run output state by each snapshot
    for (i=0; i<numberOfNodes; i++)
    {
        OpenFile (Fstate, outState, i, "state");
    } // for

    // for a simulation run output adjustment by each clock update
    for (i=0; i<numberOfNodes; i++)
    {
        OpenFile (Fadj, outAdj, i, "adj");
    } // for

    // for a simulation run output accuracy interval by each snapshot
    for (i=0; i<numberOfNodes; i++)
    {
        OpenFile (Facc, outAcc, i, "acc");
    } // for

    // for a simulation run output current rate by each snapshot
    for (i=0; i<numberOfNodes; i++)
    {
        OpenFile (Frate, outRate, i, "rate");
    } // for

    // for a simulation run output consonance by each snapshot
    for (i=0; i<numberOfNodes-1; i++)
    {
        OpenOrderFile (Fgammak, outGammak, i, "gammak");  // plots a k-th order
    } // for

    // for a simulation run output rate interval by each clock update
    for (i=0; i<numberOfNodes; i++)
    {
        OpenFile (Fdrift, outDrift, i, "drift");
    } // for
```

```
    // for a simulation run output step multiplicator by each clock update
    for (i=0; i<numberOfNodes; i++)
    {
        OpenFile (Fmult, outMult, i, "mult");
    } // for
}

// closes all plot files
void ClosePlotFiles()
{
    int i;

    Frun.close();  // close the TeX-File

    // precision
    for (i=0; i<numberOfNodes-1; i++)
    {
        Fpik[i].close();
    } // for

    // state
    for (i=0; i<numberOfNodes; i++)
    {
        Fstate[i].close();
    } // for

    // adjustment
    for (i=0; i<numberOfNodes; i++)
    {
        Fadj[i].close();
    } // for

    // accuracy interval
    for (i=0; i<numberOfNodes; i++)
    {
        Facc[i].close();
    } // for

    // current rate
    for (i=0; i<numberOfNodes; i++)
    {
        Frate[i].close();
    } // for

    // consonance
    for (i=0; i<numberOfNodes-1; i++)
    {
        Fgammak[i].close();
    } // for

    // rate interval
    for (i=0; i<numberOfNodes; i++)
    {
        Fdrift[i].close();
    } // for

    // step multiplicator
    for (i=0; i<numberOfNodes; i++)
    {
        Fmult[i].close();
    }
}

// prints the text for displaying the postscript picture 'picture'
static ostream& TexPrintPicture (ostream& out, const char *picture)
{
    out << "\\begin{figure}[h]\n";
    out << "\\epsfxsize=0.9\\textwidth\n";
```

```
    out << "\\hfill\n";
    out << "\\epsfbox{";
    out << picture;
    out << "}\n";
    out << "\\hfill\\hbox{}\n";
    out << "\\end{figure}\n\n";

    return out;
}

// print the time and date, followed by a newline
static ostream& TexPrintTime (ostream& out)
{
    out << "\\medskip\n";
    out << "\\begin{center}\n";
    out << "{\\em \\today}\n";
    out << "\\end{center}\n";
    out << "\\newpage\n\n";

    return out;
}

// prints the TeX-Output into run.tex
static void GenerateTexFile()
{
    double norm;
    char unit[8];

    ostream outRun(&Frun);
    // create the latex file to show the simulation results

    static char *buf_top =
    "
    \\documentclass[10pt,a4paper]{article}
    \\setlength{\\textwidth}{15truecm}
    \\setlength{\\textheight}{23truecm}
    \\setlength{\\oddsidemargin}{0truecm}
    \\setlength{\\evensidemargin}{0truecm}
    \\setlength{\\topmargin}{-0.06truecm}
    \\setlength{\\topskip}{0.0truecm}
    \\setlength{\\headheight}{0.0truecm}
    \\setlength{\\headsep}{0.0truecm}
    \\input{epsf}
    \\renewcommand{\\textfraction}{0.0}
    \\renewcommand{\\topfraction}{1.0}

    \\begin{document}
    ";

    outRun << buf_top << endl;

    outRun << "\\noindent" << endl;
    outRun << "{\\bf " << endl;
    switch (simutcConfigData.commonNodeData.SynchronizationAlgorithm)
    {
        case FreeRun: outRun << "Free run";
            break;
        case Midpoint: outRun << "FTA applied on midpoints";
            break;
        case FaultTolerantAverage: outRun << "FTA applied on reference points";
            break;
        case Fetzer: outRun << "FTA applied on reference points "
                            "with Fetzer for P-Nodes";
            break;
        case OP: outRun << "Optimal Precision";
            break;
        case Marzullo: outRun << "Marzullo";
            break;
        case Rate: outRun << "Rate";
```

```
        break;
    case FaultTolerantIntersection: outRun << "FTI";
        break;
    default:
        outRun << simutcConfigData.commonNodeData.SynchronizationAlgorithm;
        outRun << " (unknown)";
} // switch
outRun << "}: " << endl;
outRun << "$n = " << simutcConfigData.commonNodeData.NumberOfNodes << "$, ";
outRun<<"$P_S = " << simutcConfigData.commonNodeData.RoundPeriod << "$~s, ";
outRun << "$P_R = " << simutcConfigData.commonNodeData.RoundPeriod *
    simutcConfigData.commonNodeData.RateCycles << "$~s, ";
outRun<<"$f = "<<simutcConfigData.commonNodeData.NumberOfFaultyNodes<<"$, ";
Normalize (simutcConfigData.nodeData[0].DelayCompensation, norm, unit);
outRun << "$\\Delta = " << norm << "~{\\rm " << unit << "}$";
outRun << " \\newline" << endl;

Normalize (simutcConfigData.nodeData[0].ClockDrift, norm, unit);
outRun << "clocks:~$\\rho = (";
outRun << norm << "~{\\rm " << unit << "/s}";
for (int i=1; i<simutcConfigData.commonNodeData.NumberOfNodes; i++)
{
    Normalize (simutcConfigData.nodeData[i].ClockDrift, norm, unit);
    outRun << ", " << norm << "~{\\rm " << unit << "/s}";
}
outRun << ")$ \\newline" << endl;

outRun << "network:~$\\delta = ( ";
Normalize (simutcConfigData.networkData.Delta, norm, unit);
outRun <<  norm << "~{\\rm " << unit << "}, ";
Normalize (simutcConfigData.networkData.EpsilonMinus, norm, unit);
outRun <<  norm << "~{\\rm " << unit << "}, ";
Normalize (simutcConfigData.networkData.EpsilonPlus, norm, unit);
outRun <<  norm << "~{\\rm " << unit << "} )$, ";
Normalize (simutcConfigData.networkData.Lambda, norm, unit);
outRun << "$\\lambda = " << norm << "~{\\rm " << unit << "}$";
outRun << endl;


TexPrintPicture (outRun, "state.ps");
TexPrintPicture (outRun, "pi.ps");
TexPrintTime (outRun);

TexPrintPicture (outRun, "adj.ps");
TexPrintPicture (outRun, "acc.ps");
TexPrintTime (outRun);

TexPrintPicture (outRun, "rate.ps");
TexPrintPicture (outRun, "gamma.ps");
TexPrintTime (outRun);

TexPrintPicture (outRun, "mult.ps");
TexPrintPicture (outRun, "drift.ps");
outRun << "\\medskip" << endl << endl;


outRun << "\\noindent" << endl;
outRun << "evaluation:~rounds$_{\\max} = ";
outRun <<     evalsysConfigData.NumberOfRounds << "$, ";
outRun << "messages$_{\\max} = ";
outRun <<     evalsysConfigData.NumberOfMessages << "$, ";
outRun << "$P_{\\rm snap} = ";
outRun <<     evalsysConfigData.SnapshotPeriod << "$" << endl;
outRun << "\\vspace*{0.5in}" << endl;

static char *buf_tail =
"
\\begin{center}
{\\em \\today}
```

```
    \\end{center}

    \\end{document}
    ";

    outRun << buf_tail << endl;
}


//---------------------------------------------------------------------------
//--- sort an array of doubles, compute precision
//---------------------------------------------------------------------------

static void SortArray (double* unsortedArray, double* sortedArray, INT32 size)
{
    Assert (unsortedArray != NULL);
    Assert (sortedArray != NULL);
    Assert (size > 0);

    int i;
    int free;
    int up, low, curr;
    double item;

    free = 0;
    low = up = curr = 0;

    for (i=0;i<size;i++)
    {
        item = unsortedArray[i];

        if (free != 0)  // find the entry
        {
            up = (free>0) ? free-1 : 0;  // prepare for binary search
            low = 0;
            curr = (up+low)/2;

            while ((item != sortedArray[curr]) && (low < up))
            {
                if (item < sortedArray[curr]) up = (curr>0) ? curr-1 : 0;
                else if (item > sortedArray[curr]) low = curr+1;

                curr = (low+up)/2;
            }

            if (item > sortedArray[curr]) curr++;
            // didn't find the entry, need to insert behind the current one
        } // if

        Assert (curr < (SINT32)(size));
        Assert (curr >= 0);

        memmove (sortedArray+curr+1, sortedArray+curr,  // move everything
                 (size-curr-1)*sizeof(double));

        sortedArray[curr] = item;  // enter the timestamp

        free++;  // increment the free index
    } // for
}


static double PrecisionKth (double* pi, INT32 upper, INT32 order)
{
    INT32 lower;
    int i;

    if (upper <= 1 + order) return 0.0;
    // not enough elements
```

```
    lower = 0;
    upper--;
    // now we have pi[lower],...,pi[upper]


    for (i=0; i<order; i++)
    {
        // in each round eliminate one peripheral element
        // which one? lower or upper one that has the larger
        // distance to the next inner neigbor

        if ( (pi[lower+1] - pi[lower]) >= (pi[upper] - pi[upper-1]) )
        {
            lower++; // take away the lower
        }
        else
        {
            upper--; // take away the upper
        }

    }

    Assert( lower < upper );
    return (pi[upper] - pi[lower]);
    // compute the distance
}
#endif
```

F SimUTC/Sources/Evalsys/evalcfg.cpp

```
//
// evalcfg.cpp
//
// contains values of all globals needed for configuring the evaluation
//
// notes:
// *) to change the number of nodes or number of networks, change the
//    constants in evalcfg.hpp
//
// created: 16.09.98
//
// last change: 05.02.99
//
// RCS version control information:
//     $Revision: 1.12 $$State: Exp $$Date: 1999/02/05 15:33:51 $
//


#include "clockcfg.hpp"
#include "constant.hpp"
#include "evalcfg.hpp"


//****************************************************************************
// the variables used for accessing the data
//****************************************************************************

const SEvalsysConfigData evalsysConfigData;
const SSimutcConfigData simutcConfigData;


//****************************************************************************
// constants for the evaluation
//****************************************************************************

const double TwoPowMinus16 = 1.525878e-05;
// resolution of duty timer, approx. 15 us


SEvalsysConfigData::SEvalsysConfigData()
{
    //-------------------------------------------------------------------
    // configure how long evalsys should run
    //-------------------------------------------------------------------

    NumberOfRounds = 100; //40;
    // maximum number of rounds that are printed on screen; a round is defined
    // as the reception of an event of type RoundEvent from node RoundNodeID
    // (see below); if the constant is exceeded, the simulation is stopped
    // (killed)

    NumberOfMessages = 500;
    // maximum number of messages that are received from the simulation; if
    // the constant is exceeded, the simulation is stopped (killed)
    // note: multiply this constant with the SelectTimeout to get the number
    // of seconds evalsys will delay before killing a simulation that does
    // not send any messages


    //-------------------------------------------------------------------
    // configure from when on to collect precision data
    //-------------------------------------------------------------------

    FirstSyncPrecisionRound = 3;  // >= 0
    // evalsys traces the synchronization rounds of the simulation; it
```

```
// starts to collect precision data from the snapshots if the simulation
// is at least in synchronization round FirstSyncPrecisionRound
// note: collecting precision data only makes sense after the sync.
// algorithm has been executed; so the round should be at least the
// first round in which the sync. algorithm has been executed


//------------------------------------------------------------------------
// the area that is occupied on the nodes
//------------------------------------------------------------------------

NodeArea = 0;
// the area of the UTCSUs; currently, you can only have one area per
// nodeID


//------------------------------------------------------------------------
// nodes with special meaning (round count)
//------------------------------------------------------------------------

RoundNodeID = 1;
// the ID of the node used for counting the simulation rounds; you have
// to make sure that the node you select periodically sends the event
// you are using to count the rounds (p.e., if RoundEvent is set to
// ClockUpdate, then you should not use a P-node)

RoundArea = NodeArea;
// the area of the node used for counting the simulation rounds

RoundEvent = ClockUpdate;
// the event that causes the round counter to be incremented; only
// use events that are sent periodically by the node you have selected
// in RoundNodeID

//------------------------------------------------------------------------
// snapshot node
//------------------------------------------------------------------------

SnapshotNodeID = 2;
// the ID of the node used for triggering snapshots
// in the future (currently *not* recommended!!!):
// set it to a value >= SCommonNodeConfigData::NumberOfNodes if you do
// not want snapshots

SnapshotArea = NodeArea;
// the area of the node used for triggering snapshots

SnapshotPeriod = 3.0;//100;  // should be > 0
// the period (in seconds) of the supervisor that triggers periodic
// snapshots; this value overrides SNodeConfigData::SupervisorPeriod


//------------------------------------------------------------------------
// timeouts when waiting for simulation messages
//------------------------------------------------------------------------

BlockTime = 20;
// time (in seconds) the ReceiveMessage() function blocks when waiting
// for the header

SelectTimeout = 20;
// time (in seconds) SelectSocket() waits for one of the supervisor
// sockets to become readable


//------------------------------------------------------------------------
// general evaluation specific things
//------------------------------------------------------------------------
```

```
    MaskSize = EventCount/8 + 1;
    // size of the event mask used in SEventReport; the value should not
    // exceed the size of an INT32
}



//****************************************************************************
// constants for the simulation
//****************************************************************************

//----------------------------------------------------------------------------
// Controller configuration
//----------------------------------------------------------------------------

SControllerConfigData::SControllerConfigData()
{
    //------------------------------------------------------------------------
    // event reports (from the network)
    //------------------------------------------------------------------------

    // enabled event classes; uncomment the classes you need, comment
    // out those that you do not need
    EventClasses =
        //(1 << SystemReport) +
        //(1 << InterruptEvent) +
        //(1 << FaultInjection) +
        0;

    // enabled events; again, uncomment the events you need and comment
    // out the rest
    Events =
        (1 << CrashStatusChange) +
        (1 << DeliverMessage) +
        (1 << EarlyMessage) +
        (1 << LateMessage) +
        (1 << MessageLost) +
        (1 << SendMessage) +
        0;


    //------------------------------------------------------------------------
    // Controller period
    //------------------------------------------------------------------------

    ControllerPeriod = DefaultIfcPeriod;  // should be > 0
    // the period of the controller in seconds; this value affects how
    // fast the controller reacts to user commands and how often it sends
    // network events to the evaluation
    // in pure simulation, the value is not very important, so it is set
    // on its default value
}


//----------------------------------------------------------------------------
// node configuration
//----------------------------------------------------------------------------

SCommonNodeConfigData::SCommonNodeConfigData()
{
    //------------------------------------------------------------------------
    // number of nodes
    //------------------------------------------------------------------------

    NumberOfNodes = numberOfNodes;  // do not change this assignment!
    // number of nodes in the system
    // note: if you have to change the number of nodes, change the constant
    // 'numberOfNodes' in evalcfg.hpp
```

```
    NumberOfFaultyNodes = 0;
    // maximum number of faulty nodes


    //-------------------------------------------------------------------
    // round specific things (period, delay cycles)
    //-------------------------------------------------------------------

    RoundPeriod = 12;//100;  // should be > 0, multiple of 2^16
    // period of a synchronization round in seconds
    // the value should have duty timer granularity (2^-16 seconds)

    DelayCycles = 2;//4;  // set it to 0 to turn off delay measurement
    // every 'delayCycles'-th message does delay measurement

    RateCycles = 4;//0;  // set it to 0 to turn off rate sync.
    // every 'rateCycles'-th message sends rate synchronization data


    //-------------------------------------------------------------------
    // general settings (algorithm)
    //-------------------------------------------------------------------

    // the synchronization algorithm that is used; uncomment the algorithm
    // you want to use and leave the rest commented out
    SynchronizationAlgorithm =
        //FaultTolerantAverage;
        //FreeRun;
        //Midpoint;
        //Fetzer;
        //OP;
        //Marzullo;
        Rate;
        //FaultTolerantIntersection;
}


SSpecificNodeConfigData::SSpecificNodeConfigData()
{
    //-------------------------------------------------------------------
    // configurations concerning the clock
    //-------------------------------------------------------------------

    ClockDrift = 0;  // should be >= -1
    // clock drift
    // SimUTC does only accept the parameter in pure simulation
    // note: the ctor of SSimutcConfigData overrides the drift

    ConnectedGpus = 0;  // default: S-Node
    // the GPUs of the UTCSU which have GPS receivers connected to them
    // this is a binary representation, so if a receiver is connected to
    // GPU1, bit 1<<GPU1 is set in connectedGpus
    // this value is not directly sent to the Clock module, but for each
    // GPU set, command ConnectGps is called
    // note: the ctor of SSimutcConfigData overrides this value

    Macrostamp = 0;
    Timestamp = 0;
    // initial clock value (each is a 32-bit value)

    Accuracy = 0; //0x20000000;
    // initial clock accuracy
    // the value contains the UTCSU bits S:[-8,-38] of both accuracies
    // take care when comparing this input with the accuracies you get
    // with the clock timestamps: those are unsigned and contain
    // bits [-8,-23] in the upper and lower two bytes of the 'acc' field

    AmortizationDuration = DefaultAmortizationDuration; // should be >= 0
    // the duration of a clock amortization phase (in seconds)
```

```
// the value should have duty timer granularity (2^-16 seconds)

RhoPlus = 1e-4;//WcRhoPlus;  // should be in [0, WcRhoPlus]
// upper bound on the regular clock drift (the worst case bound for
// faulty drift is taken from WcRhoPlus)

RhoMinus = 1e-4;//WcRhoMinus;  // should be in [0, WcRhoMinus]
// lower bound on the regular clock drift (the worst case bound for
// faulty drift is taken from WcRhoMinus)


//-------------------------------------------------------------------
// configurations concerning the clock (fault injection)
//-------------------------------------------------------------------

ProbClockCrash = 0;  // should be in [0, 1]
// probability of "crashing" the clock (i.e., the clock stops counting)

ProbClockRecover = 0;  // should be in [0, 1]
// probability of recovering a crashed clock

ProbDriftChange = 0;  // should be in [0, 1]
// probability that the clock drift changes

ProbFastDrift = 0;  // should be in [0, 1]
// probability that the clock drift changes to a fast value

ProbSlowDrift = 0;  // should be in [0, 1]
// probability that the clock drift changes to a slow value

ProbReadFault = 0;  // should be in [0, 1]
// probability that a read fault occurs

ProbWriteFault = 0;  // should be in [0, 1]
// probability that a write fault occurs


//-------------------------------------------------------------------
// clock drift wobbling
//-------------------------------------------------------------------

WobblePeriod = 0;  // should be >= 0, multiple of 2^16
// the period of drift wobbling (in seconds)
// set it to zero to disable drift wobbling
// the value should have duty timer granularity (2^-16 seconds)

WobbleRhoPlus = RhoPlus;  // should be in [0, WcRhoPlus] (constant.hpp)
// upper bound on the drift during wobbling

WobbleRhoMinus = RhoMinus;  // in [0, WcRhoMinus] (constant.hpp)
// lower bound on the drift during wobbling

WobbleAddRangeMin = 0;  // should be >= 0
WobbleAddRangeMax = 0;  // should be >= 0
// bounds of the interval from which the additional drift should be
// chosen (it is taken from this interval with Uniform distribution)


//-------------------------------------------------------------------
// configurations concerning the clock synchronization module
//-------------------------------------------------------------------

DelayCompensation = TwoPowMinus16 * 120;  // >= 0, multiple of 2^16
// delay compensation (of network transmission times) in seconds
// the value should have duty timer granularity (2^-16 seconds)

CompDelayCompensation = 0;  // should be >= 0, multiple of 2^16
// computation delay compensation (algorithm execution time) in seconds
// can be set to zero in pure simulation, because the algorithm
```

```
// execution does not take any (simulated) time
// the value should have duty timer granularity (2^-16 seconds)


//---------------------------------------------------------------------
// configurations concerning the network access (fault injection)
//---------------------------------------------------------------------

ProbLanceCrash = 0;  // should be in [0, 1]
// probability of a node crash

ProbLanceRecover = 0;  // should be in [0, 1]
// probability of node recovery

DmaxSendLance = 0;  // should be >= 0
// maximum delay of outgoing messages in seconds

DmaxRecvLance = 0;  // should be >= 0
// maximum delay of incoming messages in seconds

ProbLanceSendScramble = 0;  // should be in [0, 1]
// probability of scrambling send message

ProbLanceRecvScramble = 0;  // should be in [0, 1]
// probability of scrambling receive message

ProbLanceSendDuplicate = 0;  // should be in [0, 1]
// probability of duplicating send message

ProbLanceRecvDuplicate = 0;  // should be in [0, 1]
// probability of duplicating receive message


//---------------------------------------------------------------------
// event reports (from the nodes)
//---------------------------------------------------------------------

// enabled event classes; uncomment the classes you need, comment
// out those that you do not need
EventClasses =
    (1 << SystemReport) +
    (1 << InterruptEvent) +
    //(1 << FaultInjection) +
    0;

// enabled events; again, uncomment the events you need and comment
// out the rest
Events =
    //(1 << ChangeAssociation) +
    (1 << ClockPreload) +
    //(1 << ClockReadFault) +
    (1 << ClockUpdate) +
    //(1 << ClockWriteFault) +
    //(1 << ConvergenceFailed) +
    //(1 << CrashStatusChange) +
    //(1 << DelayEstimation) +
    //(1 << DiscardRecvMessage) +
    //(1 << DiscardSendMessage) +
    //(1 << GpsEvent) +
    //(1 << IgnoreSetting) +
    //(1 << IntervalEvent) +
    (1 << RateEvent) +
    //(1 << ReceivedMessage) +
    //(1 << ScrambleRecvMessage) +
    //(1 << ScrambleSendMessage) +
    //(1 << DuplicateRecvMessage) +
    //(1 << DuplicateSendMessage) +
    //(1 << SendMessage) +
    (1 << Snapshot) +
```

```
        //(1 << TimerEvent) +
        //(1 << UserEvent) +
        0;
    // note: GPS events are only sent if the clock needs resynchronization to
    // GPS; so the event may be turned on without harm (it will only occur
    // once or twice at the beginning of the reports, until the clock is
    // synchronized to GPS


    //------------------------------------------------------------------------
    // Supervisor period
    //------------------------------------------------------------------------

    SupervisorPeriod = DefaultIfcPeriod;  // should be > 0
    // the period of the supervisor in seconds; like the controller period,
    // it affects how the supervisor reacts to user commands and how often
    // it sends CSA events to the evaluation
    // if the node triggers snapshot, SEvalsysConfigData::SnapshotPeriod
    // overrides this value
    // in pure simulation, the value is not very important

}


//----------------------------------------------------------------------------
// network configuration
//----------------------------------------------------------------------------

SNetworkConfigData::SNetworkConfigData()
{
    //------------------------------------------------------------------------
    // number of networks
    //------------------------------------------------------------------------

    NumberOfNetworks = numberOfNetworks;  // do not change this assignment!
    // the number of simulated networks
    // note: if you have to change the number of networks, change the
    // constant 'numberOfNetworks' in evalcfg.hpp
    // (always 1 in pure simulation, 0 in hardware-based simulation)
    // note: if there is more than one network, the first one is used as
    // associated network


    DefaultCINetwork = 2;
    // the associated network in hardware (is always used if numberOfNetworks
    // is zero); this should be the CI network interface number


    //------------------------------------------------------------------------
    // message transmission characteristics
    //------------------------------------------------------------------------

    Delta = 60e-6;//.6;    // should be >= EpsilonMinus (see below)
    // deterministic part of the transmission delay (in seconds)
    // first measures: d =^= 70 us

    EpsilonPlus = 20e-6;//.1;  // should be >= 0
    // indeterministic upper part of the transmission delay (in seconds)
    // eps =^= 2-3 us

    EpsilonMinus = 10e-6;//.0;  // should be <= Delta
    // indeterministic lower part of the transmission delay (in seconds)

    Lambda = 1e-3;//.2;  // should be >= 0
    // broadcast delay (in seconds)

    Omega = 0;  // should be >= 0
    // broadcast operation delay (in seconds)
    // ignored by the currently used broadcast network
```

```
    //------------------------------------------------------------------------
    // message transmission characteristics (fault injection)
    //------------------------------------------------------------------------

    Dmin = 0;  // should be in [0, Delta-EpsilonMinus]
    // minimum transmission delay of early messages (in seconds)

    Dmax = 5;  // should be >= Delta+EpsilonPlus
    // maximum transmission delay of late messages (in seconds)

    ProbNetCrash = 0;  // should be in [0, 1]
    // probability of a network crash

    ProbNetRecover = 0;  // should be in [0, 1]
    // probability of network recovery

    ProbNetEarly = 0;  // should be in [0, 1]
    // probability of early messages

    ProbNetLate = 0;  // should be in [0, 1]
    // probability of late messages
}


//----------------------------------------------------------------------------
// complete simulation configuration
//----------------------------------------------------------------------------

// the ctor modifies the default values of the nodes
SSimutcConfigData::SSimutcConfigData()
{
    int nodeNumber;  // for iterating over the nodes

    nodesDiffer = False;
    // this is the default if all nodes are configured alike
    // in this case, PrintConfiguration() in evalfunc.cpp only prints
    // the configuration of one node, not that of all nodes

    // controllerData is taken over unchanged

    // commonNodeData is taken over unchanged

    // nodeData is adapted; add your own node-specific modifications here

    nodeData = new SSpecificNodeConfigData[numberOfNodes];
    // create the default configurations

    nodesDiffer = True;  // we make changes to some nodes

    for (nodeNumber = 0;
         nodeNumber < simutcConfigData.commonNodeData.NumberOfNodes;
         nodeNumber++)
    {

        // last node is a P-Node, but only if there are at least two nodes
        if ((nodeNumber == simutcConfigData.commonNodeData.NumberOfNodes-1) &&
            (nodeNumber > 0))
        {
            //nodeData[nodeNumber].ConnectedGpus = 1<<GPU3;
        } // if

        double stop = 2.0;

        switch (nodeNumber+1)  // set clock drifts
        {
            case 0:
                break;  // do not use!!!
```

```
            // we use the real node number (1-numberOfNodes)

        case 1:
            nodeData[nodeNumber].ClockDrift = 0; //1e-6;

            // drift wobbling
            nodeData[nodeNumber].WobblePeriod = stop*TwoPowMinus16*0.0;
            break;

        case 2:
            // average drift
            nodeData[nodeNumber].ClockDrift = 0.5e-6; //0;

            // drift wobbling
            nodeData[nodeNumber].WobblePeriod = stop * TwoPowMinus16 * 5e5;
            nodeData[nodeNumber].WobbleRhoPlus = 0.3e-6 *3;
            nodeData[nodeNumber].WobbleRhoMinus = 0.1e-6 *3;
            nodeData[nodeNumber].WobbleAddRangeMin = 0.0e-6 *3;
            nodeData[nodeNumber].WobbleAddRangeMax = 0.05e-6 *3;
            break;

        case 3:
            // average drift
            nodeData[nodeNumber].ClockDrift = -0.5e-6;//-1e-6;

            // drift wobbling
            nodeData[nodeNumber].WobblePeriod = stop * TwoPowMinus16 * 3e5;
            nodeData[nodeNumber].WobbleRhoPlus = 0.1e-6 *3;
            nodeData[nodeNumber].WobbleRhoMinus = 0.1e-6 *3;
            nodeData[nodeNumber].WobbleAddRangeMin = 0.01e-6 *3;
            nodeData[nodeNumber].WobbleAddRangeMax = 0.05e-6 *3;
            break;

        case 4:  // drift wobbling
            // average drift
            nodeData[nodeNumber].ClockDrift = -1e-6;

            // drift wobbling
            nodeData[nodeNumber].WobblePeriod = stop * TwoPowMinus16 * 5e5;
            nodeData[nodeNumber].WobbleRhoPlus = 0.2e-6 *3;
            nodeData[nodeNumber].WobbleRhoMinus = 0.3e-6 *3;
            nodeData[nodeNumber].WobbleAddRangeMin = 0.01e-6 *3;
            nodeData[nodeNumber].WobbleAddRangeMax = 0.01e-6 *3;
            break;

        case 5:
            // average drift
            nodeData[nodeNumber].ClockDrift = 2e-6;

            // drift wobbling
            nodeData[nodeNumber].WobblePeriod = stop * TwoPowMinus16 * 1e6;
            nodeData[nodeNumber].WobbleRhoPlus = 0.5e-6 *3;
            nodeData[nodeNumber].WobbleRhoMinus = 0.5e-6 *3;
            nodeData[nodeNumber].WobbleAddRangeMin = 0.03e-6 *3;
            nodeData[nodeNumber].WobbleAddRangeMax = 0.03e-6 *3;
            break;
      } // switch
    } // for

    // networkData is taken over unchanged
}

// the dtor deletes the array
SSimutcConfigData::~SSimutcConfigData()
{
    delete [] nodeData;
}
```