A Software Platform Architecture for Multi-Protocol, Hardware Supported Clock Synchronization in Heterogeneous Networks

Patrick Loschmidt[1], Georg Gaderer[2], Thilo Sauter[1]
[1]Research Unit for Integrated Sensor Systems, Austrian Academy of Sciences
Viktor Kaplan Strasse 2, A-2700 Wiener Neustadt, Austria
{Patrick.Loschmidt, Thilo.Sauter}@OEAW.ac.at
[2]Institute of Computer Technology, Vienna University of Technology
Gusshausstrasse 27-29/E384, A-1040 Vienna, Austria
Georg.Gaderer@TUWien.ac.at

Abstract

In modern communication networks the task of synchronizing the system clock of nodes is of growing interest. Especially in sensor networks synchronized applications are of crucial importance, but a global view of the system status can only be examined if the individual data acquisitions are exactly related to each other. This imposes constraints on the implementation that are rarely discussed. The present work proposes a driver and protocol stack architecture which allows each node to participate in different clock synchronization mechanisms at a time. Further the important problem of hard real-time requirements within the software environment is addressed, including the critical task of detecting missed events and the continuous amortisation of system clock deviations. An insight to realisation is given with a discussion about the implementation study used in the REMPLI (Real-time Energy Management via Powerlines and Internet).

1. Introduction

With the spread of packet-oriented communication networks the demand for a shared notion of time of the participating nodes increases. The common time base for sensor and automation networks eases tasks like synchronized measurements, event triggering and relieves the media access layer of timing and collision problems [1]. Another approach is to synchronize not the communication but the applications themselves, which gives a higher-level possibility of implementing real-time [2].

Due to the fact that different applications have various demands regarding the timing accuracy, each node might participate in one or more synchronization domains concatenated in a hierarchical fashion. In such a setup, standard low accuracy protocols (e. g., Network Time Protocol [3], ∼10 ms) are used on the lower levels and are driven by other mechanisms (e. g. IEEE 1588 [4, 5], Global Positioning System [6], <100 ns) providing time information with high precision. Figure 1 gives an overview of clock synchronization methods showing the achievable accuracy and the possible network size. Considering the capabilities of the mentioned protocols, an architecture linking these technologies would be interesting.

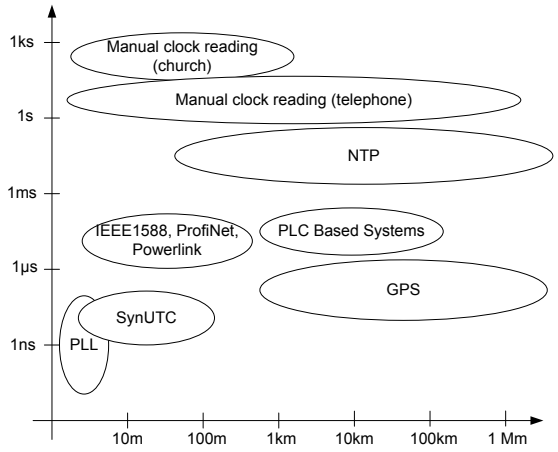


Figure 1. Clock synchronization methods, their accuracy vs. manageable size

There are various classes of nodes with respect to the way how their common notion of time is kept synchronous. A node might be client in one protocol while being master or peer in another synchronization domain. In general each system should also be capable of switching between the synchronization protocols in order to supply basic fault tolerance.

This paper proposes the implementation of a platform offering the possibility to close the gap between the different clock synchronization protocols in a heterogeneous network. Sections 2 and 3 describe the architecture and the application needs on a generic base. In the following Section 4 the concept is proven based on the imple-

0-7803-9402-X/05/$20.00 © 2005 IEEE

mentation issues in the REMPLI [7] project, where a heterogeneous synchronization between different clock synchronization protocols of the IP-world and powerline networks is needed. A conclusion and reference to further work rounds off the paper.

2. System architecture

The proposed architecture is designed to satisfy all application needs regarding clock synchronization. These demands are more than the simple correct setting of a local clock. When high accuracy is desired, also the application interface has to be considered. This can be achieved by the provision of a single signal, aligned to the absolute time, which will in the following be called *frame clock*. Additionally, the architecture aims to be extendible with all state of the art clock synchronization protocols and therefore act as a bridge between different clock adjustment methods. Although the aim of the current implementation study shown in Section 4 is an implementation based on uCLinux [8], a full-functional Linux version without MMU support, the architecture of the software platform is held as generic as possible.

The bottom layer of the proposed architecture, shown in Figure 2, is the hardware consisting of both a time slot based and a packet-oriented network interface, a high precision hardware clock, and an optional external clock reference. In order to provide high accuracy clock correction, the external reference and the network interfaces need to have a physical connection to the clock which allows triggering the generation of an event time stamp. Consequently the software is able to determine the exact time the event (e. g., reception of an absolute time information) occurred even if the corresponding piece of code is scheduled at a later point of time. A second connection from the hardware clock to the time slot based network provides the frame clock for stand-alone or master operation mode on that communication path using periodic pulses.

Owing to the high precision requirements the clock needs to support some special operations in hardware that allow the software environment to do real-time clock adjustments which are independent of the execution and scheduling time of the operating system. The already mentioned generation of event time stamps gives the necessary accuracy on incoming events whereas a periodic timer delivers a signal for triggering the time slot based interface with low skew.

One essential feature is the continuous amortisation of clock deviation by modifying the clock tick size which has to be supported in hardware. Likewise, a command sequence allowing the preparation of the modified parameters prior to actually invoking them at a certain point of time is essential for exact clock adjustments. This permits long computations of parameters whereas the actual setting in hardware at the designated apply time does not depend on any software scheduling or bus access uncertainties.

In respect to the architecture modern operating systems provide, the software platform was split into a kernel space and an user space part. While the former is responsible for accessing the hardware and ensuring correct interrupt processing, the user space applications handle the network protocol mechanisms. In order to keep the design modular and portable to other systems [9] it was tried to use only *vertical* interfacing connections.

The first software layer consists of all device drivers in the *kernel* space, handling the communication with the hardware. There is no direct communication between these drivers so that the hardware can be exchanged with other implementations (e. g., different Ethernet network cards). Time critical inter-device functionality (e. g., queueing the absolute time information of the time slot based network combined with the time stamp of the hardware clock) is done within the synchronization driver on the second software layer in the kernel. For the same reason – performance – the TCP/IP stack is typically implemented at this level.

The goal of limiting the code in the kernel space to a minimum is motivated by two reasons. First, limited functionality in the kernel allows faster response times in critical sections which is inevitably necessary for processing timing interrupts. Second, due to security reasons, library function calls are either not allowed or very limited within kernel space. This makes the programming of larger calculations (e. g., floating point operations) very difficult. Consequently, kernel space drivers mainly acknowledge hardware requests and build a queue of requests for the user space, while on the other hand processing access requests from user space to the hardware.

The *user* space applications can provide all desired functionality, even combining the information from multiple hardware devices. Because critical timing issues like interrupt acknowledgements are done in kernel space, the timing requirements are ”only” the same as for two consecutive scheduled events.

There are several tasks that are common to all protocols implemented in user space or have to be managed centrally, respectively. The main issue is to determine the protocol which is allowed to synchronize the hardware clock, if multiple protocols are available. Therefore the node configuration and synchronization management handles the selection of the primary synchronization protocol besides the task of central logging, clock quality dissemination, and a common configuration database.

2.1. Kernel space drivers

The kernel space itself is split into two layers, where all hardware drivers constitute one and the so-called *synchronization driver* the other, representing the interface layer to the user space. The first layer implements the access to the physical devices via exported function calls to the corresponding hardware driver. The synchronization driver supports one high precision hardware clock and a time slot based network interface providing absolute time in-

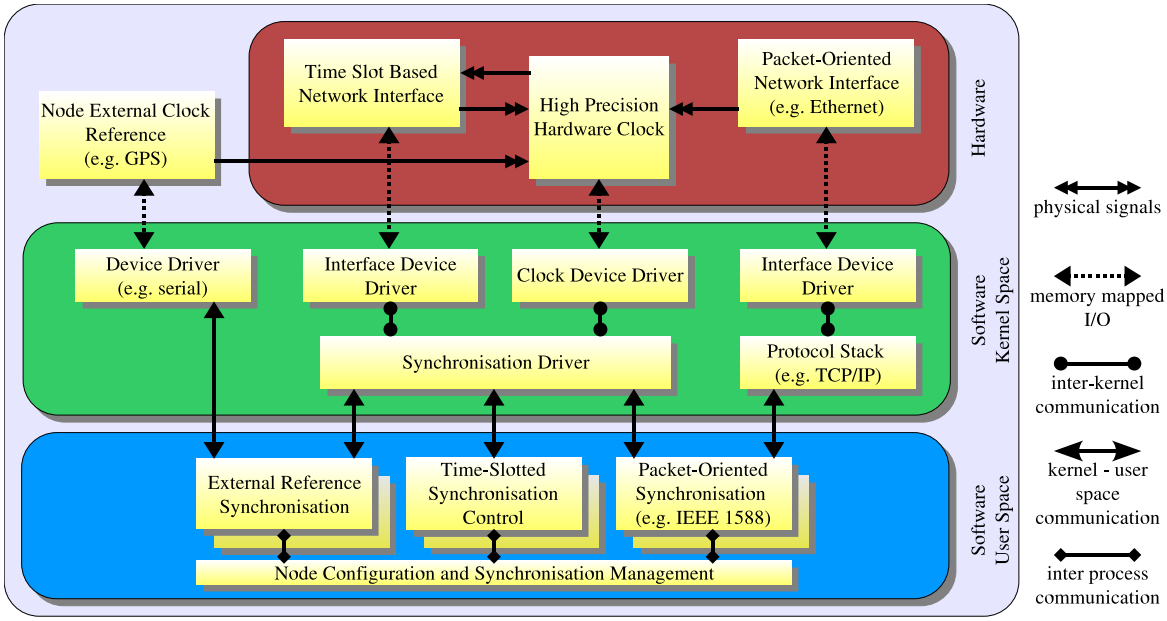


Figure 2. Proposed generic node architecture

formation in correspondence to incoming packet events. The integration of the synchronous communication path into the clock adaptation architecture allows the node to either be master or to seamlessly run in stand-alone operation in case of failures on that medium. Furthermore, the reception of absolute time information over that network provides an additional method of synchronizing the node. Consequently, the synchronization driver supports the generation and the supply of time information in correspondence to frame clock events.

Due to the fact, that the frame clock is usually in the range of or below the operating system's scheduling time (typically milliseconds) it is not reasonable to assign the task of associating the timestamp with the corresponding absolute time to the user space. The required short reaction time (within one clock period) is the reason for implementing this device-overlapping operation into the synchronization driver.

The second task of the synchronization driver is to provide a virtual device file to the user applications and to handle concurrent access to registers. Multiple accessing user processes are handled via virtual register sets. Further requirements needing special treatment arise from the task of continuous clock amortisation. This means that over a particular time interval the clock step rate is increased or decreased to avoid unsteadiness in system time. To ensure that the generated frame clock is kept constant during such a phase the synchronization driver provides virtual registers to start and stop continuous amortisation without effect on the frame clock. Details are outlined in Section 3.2.

While the user processes access the synchronization driver via a device file, the interface to the hardware clock device driver is implemented via exported function calls. It can be kept rather simple by using just the two functions *get_Register(address, data)* for read and *set_Register(address, data)* for write access to the hardware. Additionally a mechanism as described in Section 2.2 is required for interrupt support. Most user requests to the device file are just transparently passed on via an appropriate function call to the hardware device driver. Thus, the user application development can use virtual registers which behave exactly like the hardware.

2.2. Interrupt processing

The synchronization driver keeps a virtual interrupt source (denoting active interrupt lines) and a virtual interrupt enable (interrupts the application that wants to be informed) register separately for each process opening the device file. This allows each user application to request just the hardware events it requires and there is no overhead for handling interrupts intended for other processes running on the same node.

The hardware clock driver is expected to provide a callback registering function in order to supply an interrupt handling routine. This keeps the architecture modular because the hardware driver is not required to know the name of the synchronization driver's interrupt handling function and it is also possible to use different functions (not concurrently). Whenever this function is called the synchronization driver immediately copies the interrupt mask, acknowledges all pending events and checks the virtual interrupt enable register of all processes to set the appropriate bit in the virtual interrupt source register. Consequently, the processing of new interrupts is blocked only for a very short period of time and the events are "queued" for further processing by user applications.

To allow checking for unintended system status the driver also propagates interrupts not requested by any user process to all virtual registers. Additionally the driver ver-

ifies if the interrupt bit has been cleared by the application before setting it again. This debugging feature is intended to ease the discovery of timing problems within the software architecture which might occur due to scheduling effects in the operating system.

2.3. User space protocols

As mentioned above, the actual synchronization strategies, which are applied to the different communication media, are implemented as separate processes in the user space. While there is the possibility to use more than one protocol per network, every stack must have access to the hardware clock. This is necessary in order to relate the node's reception time stamp to the absolute time information gathered over the corresponding communication path. As outlined in Figure 2 the high precision hardware clock is the central and linking element between all synchronization protocols / media.

For packet-oriented networks the architecture requires a signal which triggers the generation of an event time stamp in case a synchronization packet for the node is received or sent. The transmission time stamp will tell the software when a generated packet actually left the node on the medium. Thus, a receiving node can then be informed with a so-called follow-up packet about the exact time the previous information was sent. The trigger signal should be generated at very low level in hardware in order to avoid the introduction of additional skew. In the case of Ethernet this can reasonably be done by a further device scanning the MII (Medium Independent Interface) protocol for special packet types.

The time slot based network interface is supplied with a low skew frame clock. This allows every node to take over as a communication master or to run stand-alone, which greatly improves fault tolerance. The extraction of the frame clock is then no longer a single point of failure because the information about valid time slots can also be calculated from the node's internal high precision clock.

One important thing to mention is that it is necessary to have a network-wide agreement about the frame clock phase. Due to the fact that the periodic timer is coupled to the real-time clock, it is not possible to shift the phase without changing the output frequency. Thus, the easiest solution is to have a common understanding of a point of time in the past where the first pulse was generated and to generate further pulses when the absolute time is a multiple of the period since then. This will automatically align the phase if the absolute time is synchronized.

Other time references than the above mentioned network devices are integrated via an appropriate device driver, a hardware event signal to the clock, and a corresponding external clock reference synchronization stack. The latter has, like the other protocol stacks, to fulfil the task of getting an absolute time information and fetching the matching event time stamp generated by the hardware clock. If the stack was selected by the node's configuration and synchronization management to be responsible for the hardware clock synchronization, it has to ensure correct clock amortisation as described in Section 3.2. This task can be either implemented as a library function call or as a separate process, whereas the latter would also give the possibility to collect the calculated deviations from multiple synchronization protocols and perform appropriate clock corrections.

3. Application requirements

Because all time critical tasks are performed in kernel space (either by the device driver itself or by the synchronization driver) the user space can do extensive computations on an event-triggered basis without tight timing requirements. To implement the event triggering, the proposed architecture is designed to optionally use blocking I/O communication between user and kernel space on requests to the interrupt source register of the hardware clock. This means that a program can suspend itself until a requested event (e. g., generation of a time stamp) occurs. Furthermore, the synchronization driver returns an error code if the application misses an event. Consequently it is possible to detect timing problems due to scheduling delays caused by heavy processor load.

3.1. Frame clock generation

The frame clock generation was decided to be done in hardware, but is controlled by the software environment. The use of a simple trigger (generates an interrupt at a certain point of time) would give great flexibility in terms of changes in the frequency because each external event is set by the software individually. Nevertheless, the idea was abandoned because of the high interrupt load it would generate and the risk of missing the scheduling of a trigger event on time.

The chosen solution is to use a hardware timer with adjustable period. The apparent advantage is that the periodic frame trigger generation can be done without software support. Whenever the output frequency needs to be changed, there is only the necessity for one update of the hardware register specifying the period. The clock device is expected to apply the new period immediately with the end of the actual running one to avoid an inconsistent transition. Using this method there is no need for interrupts in the normal operation mode at all. Of course in some special cases (when there is the requirement for a strict sequence, see Section 3.2) the notification about the generation of a frame clock via an interrupt is indispensable but timing is significantly more relaxed.

3.2. Amortisation phase specialities

Deviations of the high precision hardware clock are compensated by adjusting the real-time clock increment (nanoseconds per clock cycle) for a specific time period called amortisation phase. During this phase the period timer generating the frame clock would produce a wrong

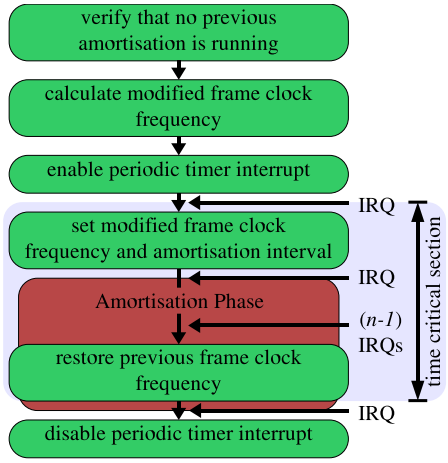


Figure 3. Sequence for hardware clock deviation amortisation

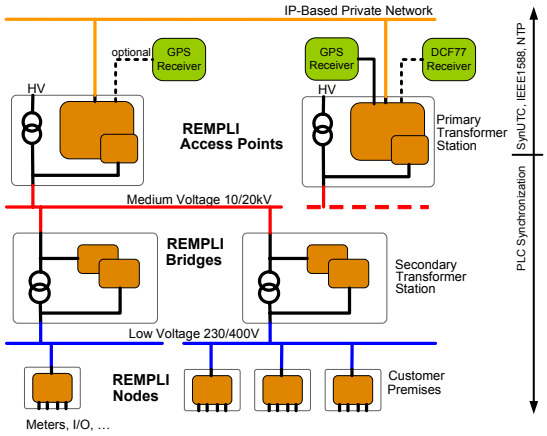


Figure 4. REMPLI architecture and clock synchronization concept

frequency at the output because it is dependent on the hardware clock running faster or slower than real-time.

The task of calculating a correct modified frequency for the amortisation is designed not to be done in hardware because it would require a division unit adding significant complexity to the integrated circuit. Furthermore, this is not even necessary because the computation can be done at any time in advance to the amortisation phase and is not time critical. Consequently the synchronisation driver assures that the frame clock period is internally compensated to hold the generated external frequency constant. To be able to do this, an amortisation phase must be calculated to be a multiple of the frame clock interval. Otherwise the period containing the end of the amortisation would vary from the desired time interval.

The initiation and running of the hardware clock correction has to follow a specific order to ensure exact timing in the hardware. The actual sequence is shown in Figure 3. During the time critical section there is a maximum command processing time of exactly two consecutive interrupts (IRQs) which is equivalent to the occurrence of the frame clock. For that reason the minimum frame clock period is dependent on the processor's computation time for the operations in the critical section, if a low skew of the frame clock output is desired. Additionally the following issues have to be considered:

- The amortisation phase has to be a multiple (n) of the current frame clock period
- The modified frame clock period is calculated by $\frac{\text{amortisation step size}}{\text{normal step size}} \times \text{frame clock period}$
- All time-consuming computations can be done outside the critical section
- Enabling the periodic timer interrupt notifies about the generation of a frame clock
- The periodic timer assigns a newly set frequency with the next frame clock

Note that this procedure is not necessary if the frame clock output is not required. In that case the duration of the amortisation phase can be chosen freely and started at any point of time.

4. Implementation case study: The REMPLI system

The REMPLI [10, 11, 7] project addresses application areas like remote meter reading and low-bandwidth user services, e.g., burglar alarm reporting. It implements a powerline communication infrastructure to connect the servers of a utility company with field-level equipment at the customers' premises.

As shown in Figure 4 the structure of the REMPLI communication network reflects the structure of the power distribution grid: Application servers are connected to access points using a private IP-based network. The latter close the gap between the IP-based network and the medium voltage (MV) network. Due to the fact that transformer stations are decoupling the medium- and low-voltage part of the power network, bridges provide transparent data connection between the low-voltage and medium-voltage power lines. From that point on, data are transmitted over the low-voltage powerline network to the nodes which connect various field level equipment such as utility meters or SCADA (Supervisory Control And Data Acquisition) equipment and thus make it accessible to application servers in the backbone network. Depending on the application requirements, REMPLI supports different standardised protocols like IEC 60870 [12], EN 62056 (also known as IEC 1107) [13], and EN 1434-3 (commonly known as M-Bus) [14].

A REMPLI access point offers the functionality to retrieve and send data to nodes connected to the REMPLI

communication infrastructure. Since the end-to-end communication takes place between application servers and nodes rather than the access points, each access point has to multiplex different data streams onto the same communication medium. While this is easy to accomplish in the IP world, where multiplexing is done by the TCP protocol, it requires additional effort in datagram-oriented field networks like PLC (Power Line Communication). Additionally, access points have the task of managing redundant communication paths. Since multiple independent access points can be attached to the same PLC network and it might happen that one access point is not able to reach all nodes, different access points have to communicate with each other in order to provide maximum possible node coverage.

The REMPLI nodes offer the interface functionality to meters, SCADA equipment, and the end user. Due to the fact, that all protocols implemented within REMPLI are used in both worlds, the node interface and the application servers, the communication between these applications need not be aware of the powerline communication in between. Since the REMPLI infrastructure covers a whole energy distribution network, an additional gain is the possibility to supervise and control the network itself. A high precision clock synchronization in REMPLI is motivated by:

- the application of the REMPLI network itself, because powerline is a highly unreliable communication medium: Since real-time communication is simply impossible in a way that delivery time can be guaranteed by means of the network layer, a strategy is to operate the network in a time-triggered fashion. Of course, classical real-time applications are – compared to high performance TDMA networks – not feasible, but logging and pre-schedulable applications are thinkable. An example for this is theft detection. In order to detect any leakage in a distribution network (e.g., water distribution networks where theft is an emerging issue or classical powerline networks, where theft is nowadays only detected by evaluation of power-consumption statistics) fine-grained loss measurements can be made by drawing the balance between utility and end user meters' values at predefined times.

- the PLC itself needs synchronized clocks which are capable of maintaining a high precision since disturbances in the powerline can cause long free run phases of nodes and bridges. To provide a fast re-establishment of the communication, the time-consuming clock recovery process on the powerline can be avoided. Moreover, due to load balancing in the energy network, the topology of the network is not stable. Thus, not only has one powerline segment to share a common notion of time, but also all access points must enable a transition of a node (or node group) from one access point to another.

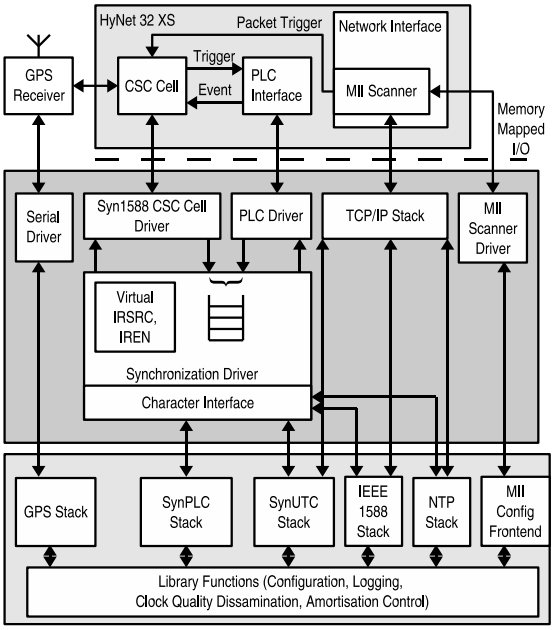


Figure 5. Schematic node implementation

Although not vital for the system design, other tasks in the REMPLI network are significantly simplified by synchronized clocks at all nodes. For example, if it is intended to change lethal communication parameters remotely, the user is faced with the fact that a wrong setting or failure can potentially end up in the need to reconfigure the nodes local and manually. An easy way to avoid that is to attach each parameter to a certain apply time and therefore ensure a backup self-reconfiguration.

4.1. Architecture protocol integration

As already mentioned in Section 2.3 all synchronization protocols are implemented in user space. This gives more flexibility in terms of available library functions and ensures fast task handling by the kernel space drivers. It is important to mention that the protocol stacks do not communicate directly with each other, but use the high precision hardware clock as a reference. Thus, the information about the absolute time is exchanged via the clock which ensures that every stack gets the maximum precision available.

Network Time Protocol (NTP)

This protocol is an example for a wide-area, fault-tolerant, but not very precise protocol. Depending on the network size the deviation of two node clocks increases with the delay of the network path between them [3]. In the example implementation a daemon (ntpd) is used to distribute the time information over the network. The coupling between the high precision hardware clock and the daemon is done via a so-called reference clock driver which translates the daemon's requests to register accesses on the clock. Due to the fact that the hardware clock is the central point of exchange the NTP stack does not have to know

about the rest of the architecture. Via the available local clock driver the daemon also serves the function of continuously adjusting the operating system clock.

IEEE 1588

The IEEE 1588 standard, which was published in 2002 [15] and adopted by IEC [16], is a solution for precise time synchronization. The so-called round-trip techniques of the protocol were originally proposed in [17, 18]. The protocol uses a master-slave topology to synchronize the distributed nodes. Using IEEE 1588, the alignment of all participating clocks in a network is done by exchanging synchronization messages between "ordinary" nodes and the elected master clock. This synchronization process takes place in two steps [19]:

First, the time difference between master and slave is appraised. This part of the synchronization process is usually referred to as offset measurement. For the appropriate correction, the master periodically transmits synchronization messages containing the estimated time the packet will leave the node to its slaves. Further, the exact time the packet was transmitted is saved. The slave clock(s) samples the time of reception, using its local clock for timestamping. Subsequently the master sends a follow-up message containing the exact point of time the previous packet was sent on to the line. Thus, the offset between sender and receiver can be calculated from the transmission and the reception timestamp. Finally, the time offset at the slave clock can be adjusted except for the network transmission delay [20].

As a second step in the protocol sequence the delay between master and slave is measured and corrected. The slave node sends a delay request and stores the point of time the packet leaves the node. After reception of this packet the master assembles a new one with the exact reception time as payload and sends it back to the slave. The difference of these two timestamps is then the network transport delay, assuming that the delay is symmetric.

The maximum achievable accuracy significantly depends on the precision of the transmit and receive timestamp. Although it is possible to implement such a timestamping as software algorithm (e.g., interrupt routines) this latency introduces an additional non-deterministic uncertainty. As a consequence, the achievable synchronization accuracy would be quite limited and only acceptable for low-end applications. Therefore, the node architecture of the REMPLI project (see Figure 5) uses a device, the MII scanner, which allows to take timestamps immediately before the packet enters the physical layer. This ensures that the variable delay introduced by the MAC (Medium Access Control) using CSMA (Carrier Sense Multiple Access) is not relevant for the time sample.

SynUTC

The approach used in SynUTC [21, 22] takes the advantage of a completely democratic clock synchronization mechanism. Therefore the critical phase after the master is lost and a new one has to be elected, is avoided. Instead of time information, data about the current accuracy interval of each clock is distributed to all nodes. Then the algorithm computes a joint interval which is the intersection of all distributed ranges and therefore smaller than the single ones [23]. Finally, each node adjusts its local clock to meet the boundaries of the newly calculated interval.

Furthermore, SynUTC introduces the so-called adder-based clock [24] which is also used for the nodes in REMPLI. Instead of a simple counter a high-resolution adder sums up a time increment on each oscillator tick. This architecture is capable of dealing with any oscillator frequency and synchronization can be done in a continuous amortisation fashion, without any unsteadiness in system time.

With the help of the adder-based clock, not only the states of the clocks in the system are synchronized, but also their progression. Additionally, since only the progression of the clocks is adapted, the clock time can be observed as a continuous function of the real-time – as opposed to hard synchronization approaches where the clock is simply set anew from time to time.

Power Line Communication (PLC)

As already mentioned, the time-variant network topology of a powerline network leads to the problem that any node which is transferred from one hierarchical clock domain to another needs to be synchronized in its clock state and clock rate, respectively. The latter is needed due to the fact that nodes need to logon into their new domain without a time-consuming clock recovery process.

Synchronization under these prerequisites can be done in a hierarchical style according to the network topology (Figure 4). The top level takes advantage of the high performance network. Between these powerline access points a clock synchronization strategy like IEEE 1588 or SynUTC can provide consistent clock states. The hierarchy below is updated by these access points in a master-slave fashion [25]. Bridges, which have in the REMPLI project the task of acting as a slave to the mid-voltage side and transmitting the messages as master to the low-voltage side are physically built as two REMPLI nodes. Synchronization between the nodes within the bridge is done via a physical clock line, where an additional datagram transmission delivers the absolute time to each rising edge of the clock line. The same concept is used at the nodes, since the PLC modem is able to deliver the receiving frame clock.

One characteristic of the used PLC technology is the fact that not all nodes are always reachable by the access points and bridges directly. Therefore every node has to act as a repeater. This has to be reflected by the clock synchronization concept, since every repeater hop degrades the clock accuracy by increasing the jitter. To be able to correct this, the hop count is also maintained by the

synchronization stack in order to estimate a local accuracy. This accuracy can be taken as a decision criterion for the absolute time if a node is synchronized by two access points, which do have in general a different hop count reaching one node.

5. Conclusion and further work

Clock synchronization in heterogeneous networks is an issue which requires a specialised concept like the one outlined in this paper. One basic issue of the software platform is the support of multiple synchronization protocols, which needs to be reflected in the architecture. With the help of such a multi-protocol support heterogeneous networks can be synchronized, the software platform acting as a bridge between the protocol worlds. Indeed this is a requirement because the possibility to use a clock synchronization protocol highly depends on the physical layer itself. For example, state-of-the-art protocols like IEEE 1588 cannot be easily applied in powerline networks, although a synchronization between different, not directly connected PLC networks is needed. This paper shows an implementation study for SynUTC, IEEE 1588, NTP, and PLC synchronization as a proof of concept. Further research issues are the broadening of the concept to gateways synchronizing fieldbus and industrial Ethernet networks as well as implementation studies of other clock synchronization concepts.

References

[1] Paolo Verissimo and Luis Rodrigues. *Distributed Systems for System Architects*. Kluwer Academic Publishers, 2001.

[2] Hermann Kopetz. *Design principles for Distributed Embedded Applications*. Kluwer Academic Publishers, 1997.

[3] David L. Mills. Internet time synchronization: the network time protocol. In *IEEE Transactions on Commmunications*, volume COM-39, pages 1482–1493, October 1991.

[4] Hans Weibel. Uhren mit IEEE 1588 synchronisieren. *Bulletin SEV/VSE*, page 35, 2004.

[5] *1588 IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems*. IEEE Instrumentation and Measurement Society TC9.

[6] Peter H. Dana. Global Positioning System (GPS) time dissemination for real-time applications. *Real-Time Systems*, 12(1):9–40, January 1997.

[7] REMPLI BIG. Real-time energy management via powerlines and internet (http://www.rempli.org).

[8] uClinux™– Embedded Linux Microcontroller Project – home page (http://www.uclinux.org).

[9] Sergio Perez and Joan Vila. Building distributed embedded systems with rtlinux-gpl. In *Proceedings of the 2003 Conferecne on Emerging Technologies and Factory Automation*, pages 161–168, 2003.

[10] Maxim Lobashov, Gerhard Pratl, and Thilo Sauter. Implications of power-line communication on distributed data acquisition and control system. In *Proceedings of the 2003 IEEE Conference on Emerging Technologies and Factory Automation*, pages 607–613, September 2003.

[11] Gerhard Pratl and Maxim Lobashov. Remote access to power-line networked nodes: Digging the tunnel. In *Proceedings of the 2004 IEEE International Workshop on Factory Communication Systems*, pages 323–332, September 2004.

[12] *IEC 60870-5-x Telecontrol equipment and systems - Part 5: Transmission protocols*. IEC TC57, 1990.

[13] *EN 62056-21 Electricity metering - Data exchange for meter reading, tariff and load control*. CENELEC IEC TC13, 2002.

[14] *Communication systems for meters and remote reading of meters - Part 3: Dedicated application layer (M-Bus)*. CEN TC 176.

[15] *1588 IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems*. IEEE Instrumentation and Measurement Society TC9, 2000.

[16] IEC. *Precision clock synchronization protocol for networked measurement and control systems*. IEC, 2004-09.

[17] Christof Fetzer and Flaviu Cristian. An optimal internal clock synchronization algorithm. In *Proceedings 10th Annual IEEE Conference on Computer Assurance*, Gaithersburg, MD, jun 1995.

[18] Flaviu Cristian. Probabilistic clock synchronization. *Distributed Computing*, 3(3):146–158, 1989.

[19] Jürgen Jasperneite, Khaled Shehab, and Karl Weber. Enhancements to the time synchronization standard ieee-1588 for a system of cascaded bridges. In *Proceedings of the 2004 IEEE International Workshop on Factory Communication Systems*, pages 239–244, September 2004.

[20] John C. Eidson and Kang Lee. Sharing a common sense of time. *IEEE Instrumentation and Measurement Magazine*, 6(1):26–32, 2003.

[21] Martin Horauer, Ulrich Schmid, and Klaus Schossmaier. NTI: A Network Time Interface M-Module for high-accuracy clock synchronization. In *Proceedings 6th International Workshop on Parallel and Distributed Real-Time Systems (WPDRTS'98)*, pages 1067–1076, Orlando, Florida, March 30 – April 3 1998.

[22] Ulrich Schmid. Synchronized Universal Time Coordinated for distributed real-time systems. *Control Engineering Practice*, 3(6):877–884, 1995. (Reprint from Proceedings 19th IFAC/IFIP Workshop on Real-Time Programming (WRTP'94), Lake Reichenau/Germany, 1994, p. 101–107.).

[23] Klaus Schossmaier. *Interval-based Clock State and Rate Synchronization*. Dissertation, , Faculty of Technical and Natural Sciences, 1998.

[24] Martin Horauer. *Clock Synchronization in Distributed Systems*. PhD thesis, Vienna University of Technology, 2004.

[25] Georg Gaderer, Thilo Sauter, and Gerd Bumiller. Clock synchronization in powerline networks. In *Proceedings of the 2005 IEEE International Symposium on Power Line Communications and its Applications*, pages 71–75, 2005.