A Self-Aware Clock for Pervasive Computing Systems

Andrea Bondavalli, Andrea Ceccarelli, Lorenzo Falai
University of Florence, Viale Morgagni 65, I-50134, Firenze, Italy
bondavalli@unifi.it, 3697838@student.unifi.it, lorenzo.falai@unifi.it

Abstract

The paper addresses the challenges and opportunities of instrumenting pervasive computing systems with a logical clock, aware of the quality of synchronization with respect to a time reference. Pervasive computing systems are i) mobile, ii) dynamic and iii) composed of a large number of distributed components; in systems with these characteristics the availability of a "smart" clock that is i) capable to use different mechanisms for the synchronization with the global distributed time reference and ii) aware of the current quality of synchronization with such time reference, can be very useful in order to build dependable middleware services and applications.

1 Introduction

Distributed pervasive systems, that are systems composed by a large populations of connected and cooperating nodes, may be highly dynamic (especially in wireless context), because of nodes or groups of nodes that can suddenly join or leave the distributed system. Nodes of pervasive systems can be very different one another, in fact they can use different software (e.g. a different synchronization mechanism), different hardware (e.g. they can have a GPS receiver or not), different connections and, finally, nodes can be of different types (sensors, embedded automotive processors, embedded roadside processors, ...). An important requirement for pervasive computing systems is the ability to adapt at runtime to handle user mobility, changing user needs, varying resources and system faults. For example, in Wireless Sensor Networks (WSNs, [1]) application requirements may vary over time and are in general not predictable, since they depend on the sensed phenomena: choosing and tuning the sufficient and necessary level of synchronization is a non-trivial problem, and to some degree the application requirements on time synchronization must be specified at design-time. However, the dynamics of the application and the environment are likely to dictate that automatic adaptation at run-time is also necessary.

A high number of distributed applications running in unpredictable or unreliable environments (e.g. Internet-based pervasive systems, or mobile ad-hoc network) have timeliness requirements. In some real-time applications meeting timing constraints is important, but it is acceptable to occasionally miss some of them, if the most important ones are achieved. As already stated, this 'indulgence' by the system designer derives from the impossibility of determining a unique bound for those constraints: in distributed dynamic environment it is often difficult to evaluate a worst-case scenario, or often the worst-case obtained is very far from the normal case ([14]).

The great need for time-dependent protocols has increased the attention about clock synchronization algorithms: starting (usually) from the creation of a local virtual clock at each node of the distributed system, communication between nodes allows to execute internal (e.g. TTA [7], developed for embedded applications), external (e.g. NTP [9], a hierarchical/master-based algorithm) or hybrid internal/external (e.g. CesiumSpray [17]) clock synchronization protocols.

In large scale systems, e.g. Internet, hierarchical or master-based algorithms are preferred, since the high number of nodes and the distance among them make the fulfillment of a cooperative algorithm very hard. On the other hand, in WSNs global time-scales and a-priori synchronization mechanisms (that is, clock are pre-synchronized when an event occurs) often are not the right choice, because of the requirements ([5]) of energy efficiency (energy spent synchronizing clocks should be as small as possible), scalability (large population of sensor nodes must be supported), robustness (the service must continuously adapt to conditions inside the network, even in case of network partitioning) and ad hoc deployment (time synchronization must work with no a priori configuration, and no infrastructure available).

All cited systems, protocols and algorithms show a consensus about the fact that quality of clock synchronization is a variable factor, very hard to predict. Many causes may be responsible for variations such as varying communication delays, propagation delays, inaccessibility of reference nodes, network partitioning, node failures, or any possible kind of failure in the clock synchronization algorithm or in the clocks itself.

Applications in pervasive systems that require clock synchronization and timeliness requirements can take advantage from the awareness of the quality of synchronization. For example, let us consider distributed ordering algorithms: good quality of synchronization of nodes can improve the ability to order events and to establish precedence between distributed events.

Another example relates to distributed durations in a quasi-synchronous system, where different bounds on transmission delays exist. In a classical system, there is usually a worst-case bound on transmission delay, and a worst-case bound on system precision (that is, difference between clocks is surely within the given bound, but this is the unique bound considered, even if the effective quality of clocks is much better than it). In a pervasive system, for the reason previously given, a unique bound is often a value very far from normal case, and different applications may have different requirements and bounds. Let us consider the quasi-synchronous system model ([14]): transmission delay is no more a parameter that must be within a unique value $max \quad T_D$, but there are other possible (shorter) bounds (however, the $max \ T_D$ bound still holds and it is the worst-case bound). We can suppose that, if there is the possibility to adopt several bounds on transmission delay, there must be the possibility to use several values of system precision, better than the assumed lower bound. A better estimation of system precision than simply using the worst case bound allows to compute more accurate time measurements and to decide more accurately the bound of the measured transmission delays (see Section 4 for a more detailed example).

In this paper we propose a light, highly-portable, low-intrusive oracle of the quality of clock synchronization. It allows to keep the the nodes of a network aware about the quality of synchronization node-to-node and node-to-time reference. Our self-aware clock (here after called SAClock) can help in WSN ([5]), Vehicular Ad hoc NETworks (VANETs, [12]) and Car-to-Car (C2C) applications, both for i) elastic/adaptive applications (real-time mission-critical applications, soft real-time applications, as in [3]), which are able to relax timeliness requirements (and time synchronization requirements) and continue to work properly in a degraded situation, and for ii) non time-elastic applications that, because of the strict real-time requirements, need an off-line low-intrusive high-dependable synchronization monitor.

The paper is organized as follows: in Section 2 we explain the context, the related problems and how we propose to solve them. In Section 3 we describe the architecture and interface of SAClock component, Section 4 describes some applications and algorithms that benefit of the SAClock and Section 5 contains our conclusions.

2 Context, Motivations and Rationale

In this section we explain the context in which our SAClock is used, the motivations that lead to the development of the component, and an high-level explanation.

2.1 Context: Pervasive Systems and Time

In distributed, open, dynamic pervasive systems, applications may have to deal with critical aspects, as temporal order delivery (for example, the physical time of sensor readings in data fusion process), and reduced and reliable transmission delay. In wireless networks, services require strict dependability, security and real-time requirements, accounting for both accidental and malicious faults in networks of mobile devices.

These applications are time-dependent. Timestamps can be obtained by reading the local clocks of the nodes of the distributed system (the most common models assume that each node has a local physical clock with bounded drift rate). Time measurements can be obtained through these timestamps. Clock synchronization is thus a fundamental process; in pervasive (usually asynchronous) systems it is impossible to ensure it a-priori. Clocks, due to their imperfections, drift from real time, and their drifts may vary over time due to several causes. There is thus the need of synchronizing the clocks to each other or to a time reference, in order to enforce and maintain accuracy and precision bounds [13]. Usually systems ([4], [14]) assume worst-case bounds to allow distinguishing unreliable biased data due to poorly-synchronized clocks, from reliable data collected when clock synchronization is good. However these bounds are usually pessimistic values, far from the normal case.

More formally, the behavior of a local clock can be described defining the three quantities *precision*, *accuracy* and *drift*. Precision π describes how closely local clocks remain synchronized to each other at any time, while accuracy α_i describes how local clock of the process P_i is synchronized at any time to a real time reference, provided externally; accuracy is thus an upper bound to the distance between local clocks and real time. As a consequence of these definitions, considering for example a set of two clocks respectively with accuracy α_1 and α_2, precision is at least as good as $\pi = \alpha_1 + \alpha_2$. Instead, if we have only system accuracy α (that is, a bound on the distance of the clocks of the distributed systems from the time reference), system precision can be estimated as $\pi = 2\alpha$. Note that accuracy is meaningless if a time reference is not defined. Finally, drift describes the rate of deviation of a clock from the time reference. The clock synchronization can be defined as the process of maintaining the properties of precision, accuracy and drift of a clock set ([18]). Figure 1 exemplifies the concepts of precision π, accuracy α, drift and clock synchronization; the outside thick dashed lines represent the bound in the rate of drift, a fundamental assumption for deterministic clock synchronization, since it allows to predict the maximum deviation after a given time interval.

Clock synchronization is a fundamental process, but in pervasive (usually asynchronous) systems it is impossible to ensure it a-priori. Instead, in these systems there are several threats to synchronization quality, and these threats are often related to the heterogeneous and dynamic nature of pervasive systems: as example of pervasive computing systems we can think to futuristic C2C (Car to Car) applications (HIDENETS [6]). A way to obtain high degree of synchronization for these applications is to use a GPS receiver at each node: this often allows a perfect and immediate receiving of UTC data. Many services are simple to build using the abstraction of a perfect distributed clock. However, even using GPS we can encounter some problems in the quality of the synchronization: for example, GPS signal can temporarily be unavailable due to buildings/tunnels or failures; some cars may not have a GPS receiver and some others may have GPS receiver of bad quality. These situations can lead to a biased local vision of UTC.

As another example, in WSNs GPS cannot be used, because it is too expensive in term of energy consumption and component costs, since it needs high-performance digital signal processing capabilities.

Moreover, physical clocks are not perfect and they continuously drift from real time, and their drift may vary over time due to causes such as temperature variation; drift can be very big, especially in WSN, in which the hardware used is usually cheap and thus of low quality.

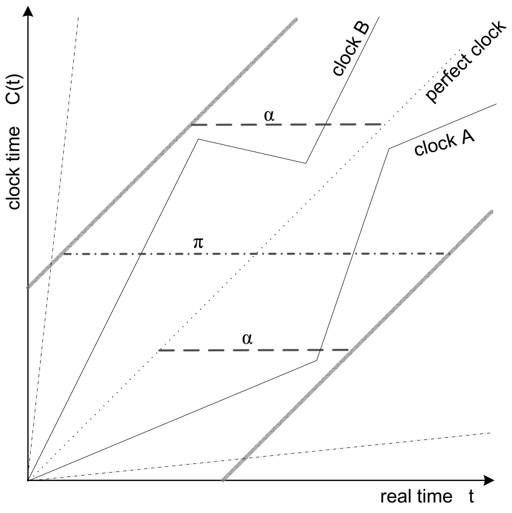

Figure 1. A set of two clocks: synchronization, accuracy, precision and drift

Summarizing, the threats to clock synchronization in pervasive system are related to i) the dynamic characteristics of groups of nodes (density of traffic in a C2C communication system, as VANETs, may impact the maximum number of hops available for information exchange, consequently increasing the transmission delay: in some synchronization algorithms the shortest is the transmission delay, the best is the clock offset estimation, as shown in [10]); ii) the environment itself (GPS requires a line of sight to the GPS satellite, which can be temporarily not available due to buildings or tunnels); and iii) the hardware used in the node itself (e.g.: the quality of the oscillators used to build physical clocks and the quality of the GPS receiver used).

We can conclude this brief overview by stating that all the cited groups of applications and systems are characterized by timeliness assumptions that may be violated, producing timing failures. An analysis of the effect of timing failures on application correctness ([16], [15]) shows that timing failures may lead to unexpected delay, with two undesirable side effects: i) a long-term one, of decreased coverage of assumptions and ii) an instantaneous one, of contamination. Contamination is defined as the incorrect behavior resulting from the violation of safety properties due to the occurrence of timing failures. Coverage is the degree of correspondence between system timeliness assumptions and what the environment can guarantee: whenever we design a system under the assumption of the absence of timing failures, we have in mind a certain (assumed) coverage. In a system with uncertain timeliness, this correspondence is not constant and it varies during system life: we call this phenomena decreased coverage. If the environment conditions start degrading, the coverage incrementally decreases and the probability of timing failure increases; on the contrary, if coverage is better than assumed, we are not taking full advantage from what the environment gives.

Dealing with these effects requires some capability of acting timely at critical moments. Here we can distinguish between ([3]) i) time-elastic applications and ii) non time-elastic applications (an example of non-time elastic applications are hard real-time applications: in these applications any failure to meet timeliness requirements may have a high cost associated [18]). In this paper we focus on time-elastic applications.

Applications that belong to the time-elastic class are able to adapt timing constraints during execution. If the worst-case is far from the normal case, the application may want to make a tradeoff between timeliness and timing faults by adapting the assumed worst-case. This raises the problem of handling timing failures: it is necessary to provide timing failure detectors. For example, an immediate effects of a timing failure is unexpected delay, defined as the violation of a timeliness property: this can be accepted if applications are prepared to work correctly under increased delay expectation. This concept relates to the possibility for an application or a system to adapt their timing expectations to the actual conditions of the environment, possibly sacrificing the quality of other (non time-related) parameters. Examples of these applications are ([18]) real-time mission-critical applications and soft real-time applications. Mission critical systems are complex systems and/or systems where the environment behavior is not totally specified, such that timeliness requirements cannot be fully guaranteed: any failure to meet timeliness requirements may have a cost associated, but the occasional failure to meet those requirements is considered an exception. Soft real-time systems are systems where occasional failure to meet timeliness requirements is acceptable.

2.2 Motivations: Interaction Between Clock Synchronization Mechanism and Applications

Timeliness properties and clock synchronization are fundamental elements in the distributed pervasive environment: clock synchronization is essential in fulfilling the timeliness requirements for these systems, and trying to estimate the achieved quality of synchronization can benefit distributed applications. In this section we focus on the interactions between clock synchronization mechanisms and applications, and on the accuracy and precision requirements that applications in pervasive systems have.

Some synchronization algorithms do not communicate to the applications the quality of synchronization, and they do not have the ability to set an upper bound of the synchronization quality and to signal when this upper bound is violated. On the other hand, other synchronization algorithms (like [8]) allow to set an upper bound on the synchronization quality; these algorithms are often used in real-time distributed systems (e.g.: embedded distributed control systems).

System models, such as the quasi-synchronous [14] or the timed asynchronous [4], usually assume the presence of a know worst-case bound for precision and accuracy, but often this value is far from the normal case or from the requirements of some applications. Different applications on the same node may require different accuracy levels, and in some cases, especially if high-quality hardware (like GPS receiver) is not available, the upper bound established would be too large for most applications, and can decrease the performance of the applications.

Traditional synchronization protocols try to achieve the highest degree of accuracy. But the level of accuracy that an application needs depends on the application requirements, and it is not an a-priori value for the whole system: therefore, there is a need for a trade-off between resource requirements and accuracy, depending on the need of the application and resource availability of the system. This situation fits perfectly in WSNs context, where there is a great attention on the resource usage (usually, the higher is the level of accuracy required, the higher is the resource requirement). In WSNs clock synchronization might not be necessary at all time (e.g. it can be needed only during sensor readings integration): providing clock synchronization all the time is a waste of the (limited) available resources. The sensor clocks can be allowed to go out of synchronization, and then re-synchronize only when needed, thereby saving resources. For example, in [11] a external/internal probabilistic synchronization algorithms which allow to provide a probabilistic bound on the accuracy of the clock synchronization, allowing for a tradeoff between accuracy and resource requirement, is proposed (in situations where the system energy is extremely low, it might not be possible to provide high accuracy).

In open, large scale, mobile pervasive systems different nodes can use different synchronization mechanisms. Despite this, applications running on the nodes may have clock synchronization requirements to be accomplished in order to deliver proper service. Moreover, a single node on a pervasive system can use, during its life, different synchronization mechanisms (as an example, a node can use a GPS hardware to receive UTC time directly from reference clocks, then, if the GPS breaks or it cannot connect, the node can connect to an NTP primary or secondary server using a wireless connection). Or it can use the same synchronization mechanism in different conditions, achieving very different synchronization quality. Or, as already seen in WSNs context, it can decide to reduce communications due to energy management requirements, shutting down the synchronization mechanisms.

Different applications on the same node may have different accuracy and precision requirements. Thus, an approach in which we just put a unique bound on clock accuracy is not appropriate if we do not want to limit any of the applications: some applications may require high clock accuracy, while other applications may require lower clock accuracy bounds.

Information about current clock synchronization quality can be used to provide a continuous estimation of system precision, in order to allow the various applications to decide if their requirements of system synchronization are met or not.

2.3 Rationale: a General Discussion on the SAClock

We propose here a logical self-aware clock, called SAClock hereafter, which allows the applications to know continuously how well the local clock of the nodes is synchronized to real time and to other clocks. In other words, the SAClock works like an oracle able do deliver continuously reliable information about accuracy of the clock with respect to real time.

The SAClock hides to applications the existence of the synchronization mechanism that the SAClock is monitoring: it just shows to the applications the current time value and the achieved quality of clock synchronization.

Our self-aware clock is composed of two parts: i) an interface, which is common to all applications and processes to guarantee portability and which just allows to interact with the SAClock, and ii) a lower level, which contains the implementations of the SAClock. This second part of the service may need different reimplementations to adapt to different synchronization mechanisms, hardware, operating systems and networks.

The SAClock can be built in two different ways, depending on the objective for which the SAClock is used: i) the objective is to maintain clock accuracy within a lower bound, or ii) there is not just a single lower bound of accuracy, and the system that can work in degraded mode depending on the value of accuracy.

In the first case, the SAClock queries the synchronization mechanism to get accuracy and, if the computed accuracy is close to the bound (or it is greater than the bound), the SAClock can force the synchronization mechanism to synchronize (if the synchronization mechanism implements procedures that allow an external source to force a new synchronization) or can impose the system to switch to another synchronization mechanism (if available).

In the second case, the SAClock queries the synchronization mechanism to get local clock accuracy and informs the applications about the local clock accuracy. Applications that are able to take advantage from the awareness of the synchronization quality can successfully use the information received from the SAClock.

SAClock implements two primitives that applications can call: *getTime()* and *setBound()*. The primitive *getTime()* allows the applications/middleware services to get both the current time and an upper bound of the estimated distance of the clock from real time; the primitive *setBound()* allows an application to set a bound on accuracy.

The SAClock interacts with the synchronization mechanism and queries it at specific time intervals to obtain information necessary to determinate accuracy values. In order to limit the query frequency to the synchronization mechanism, it uses the local clock drift to provide an estimation of local clock accuracy. Thus, the SAClock works as a 'caching' service of the synchronization mechanism: a call to *getTime()* does not imply a query to the synchronization mechanism. This caching service allows the SAClock to provide reliable information about accuracy even if the synchronization mechanism is currently not available (for example, because the synchronization mechanism is shut down, or because of a connectionless period).

The caching service allows the system designer to implement the following three strategies: i) the system designer can instantiate a polling strategy of the SAClock to the synchronization mechanism: at specific time intervals, the SAClock queries the synchronization mechanism to get current accuracy, and uses an accuracy forecaster to predict how accuracy varies, ii) if the synchronization mechanism is not working, the SAClock can forecast accuracy, and so it can continue to guarantee to applications accuracy bounds better than the worst-case one, iii) if the system switches to another clock synchronization mechanism, the SAClock can provide accuracy value even during the startup (transient) phase of the new synchronization mechanism, in order to continue to guarantee to applications accuracy bounds better than the worst-case bound (this situation is close to the previous one).

In Figure 2 a global overview of the behavior of the SAClock and the interactions between different SAClocks on different nodes is shown. Different nodes can use different clock synchronization mechanisms (and more than one clock synchronization mechanism may be available on each node). The application that lean on the SAClock can be on any level: for example, they can be high level applications or middleware services or services related to the operative system.

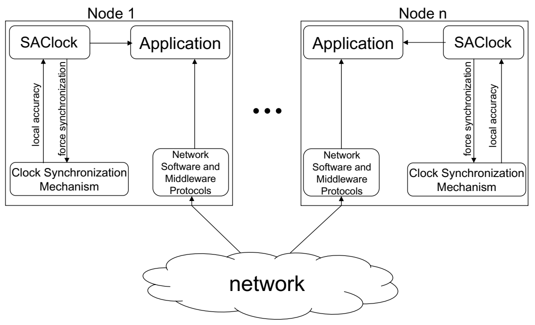
Node 1
SAClock
Application
local accuracy
force synchronization
Clock Synchronization Mechanism
Network Software and Middleware Protocols
Node n
Application
SAClock
force synchronization
local accuracy
Network Software and Middleware Protocols
Clock Synchronization Mechanism
network

Figure 2. Interactions between nodes that implement the SAClock

3 The SAClock

In this section a brief description of the interface of the SAClock and a global overview of how it works (see Figure 3) is given. The description is independent from the programming language, operating system and synchronization mechanism used.

3.1 Monitoring clock synchronization quality

The SAClock allows to estimate local clock accuracy to global time, and applications can get both time and current accuracy by the primitive *getTime()*. Applications that have accuracy requirements can use the SAClock primitive *setBound()* to set a new accuracy bound. Note that *setBound()* can be called several times.

The SAClock queries the synchronization mechanism to get an actual estimation of the local clock accuracy (through the procedure *getAccuracy()*). When an application wants to know the current time or the current accuracy (or both), it can use the procedure *getTime()* to query the SAClock. This primitive returns a tuple $< ts, acc >$, where *ts* is the value of the local clock and *acc* is the accuracy on this value.

Once the SAClock has queried the synchronization mechanism to compute accuracy, the accuracy is estimated using an accuracy forecaster, without further queries to the synchronization mechanism. This mechanism forecasts the local clock behavior: it uses information drift to control if the clock accuracy is within a given bound; only when accuracy gets close to the stated limit then more accurate time information are required to the clock synchronization subsystem (and so, there is a new call to *getAccuracy()*).

The system designer can set a default value for the clock drift, probably the worst-case bound on clock drift (see Section 2): if the synchronization mechanism in use provides better information about the actual clock drift, the SAClock can query the synchronization mechanism (through the procedure *getDrift()*) to get a drift computed at run-time and then uses it instead of the default value.

The SAClock may have the ability to force a synchronization, or to impose to the system to change the synchronization mechanism in use. If accuracy is not sufficient, *forceSynchronization()* can be used to impose the synchronization mechanism to make an attempt to synchronize, or to force the system to switch to another synchronization mechanism (if available).

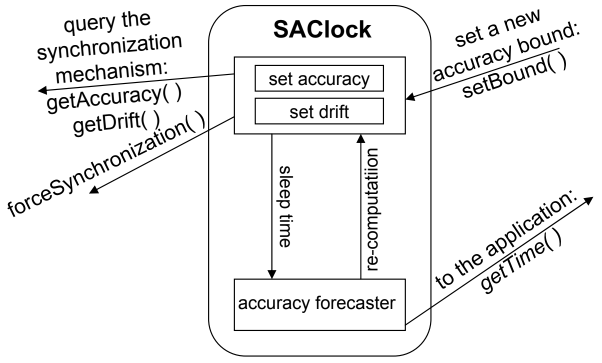

Figure 3. An high level overview of the SAClock

4 Examples of Services that Can Benefit from SAClock

In this section we give some examples applications and algorithms that can take advantage of the use of the SAClock.

4.1 Distributed System Precision

Let us consider Figure 4. Two events (on two different nodes) happen: the uncertainty in inserting the events on the time line is strictly related to the synchronization quality of the local clocks. Thus, the accuracy of the clocks of the involved nodes is the fundamental parameter to decides about the uncertainty in timestamped events.

When pairs or groups of nodes are communicating, some distributed applications may require to know the precision of the distributed system composed of the communicating nodes. If every node of the subsystem implements the SAClock, the precision can be estimated using the accuracy of the nodes of the distributed system. For example, system precision can be estimated as the sum of the two worst accuracies of the nodes of the distributed system.

This precision estimation allows to accomplish two different tasks: i) to state how good was synchronization between nodes; and ii) to separate measurements/events/operations collected and executed

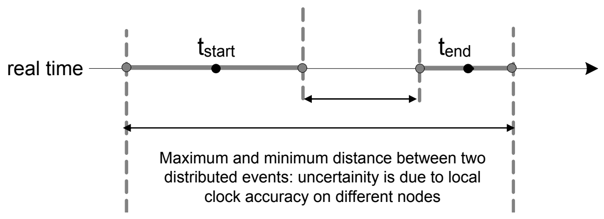
t_{start}
t_{end}
real time
Maximum and minimum distance between two distributed events: uncertainity is due to local clock accuracy on different nodes

Figure 4. Uncertainty in distance between distributed events

while clocks are poorly synchronized from measurements/events/operations collected and executed in situations of good synchronization.

Precision may change over time, as the quality of synchronization of the clock involved in the distribute system changes. For example, let us consider the system composed of two nodes shown in Figure 1, and let us suppose to compute the accuracy of each clock as the actual distance between the clock itself and the perfect clock. If precision is computed as the sum of the two local accuracies, it is no more a static value and it is highly influenced by local clock synchronization quality.

4.2 Quantitative Evaluation of Distributed Measurements

In the experimental quantitative evaluations of distributed algorithms the majority and usually more interesting metrics to analyze are distributed durations, that are intervals of time whose extremes are events local to different parts of the systems. Since local clocks are used to timestamp events, the quality of the clock synchronization has a high impact on the faithfulness of the time measurements obtained through these timestamps. Since results of analysis depend on time measurements, it is important to select reliable trusted measurements, and separate them from biased data, that could lead to wrong conclusions.

Figure 5 depicts a simple example of the uncertainty in time measurements: excessively poor clock synchronization makes it impossible to get a meaningful and accurate measurements of distributed time intervals (α_1 is the accuracy of the clock of the sender node, while α_2 is the accuracy of the clock of the receiver node).

It is important to have methods to control the level of synchronization of the processes composing the distributed systems, and thus methods available to check the accuracy of the individual measurements of a distributed metric: controlling the local clocks behavior allows to increase the quality and reliability of collected data, in fact measurements collected while clocks are poorly synchronized are split from measurements collected in situations of good synchronization, preventing the results of the analysis to be affected by unreliable biased data. Using a SAClock on each node of the distributed system allows to evaluate local clock accuracy, and, from accuracy, system precision. The quality of a distributed time measurement is evaluated considering not the distributed system precision, but considering only the precision of the clocks of the nodes that are involved in that distributed time measurement (thus, we obtain more accurate values to precision, and consequently a better estimation of quality of the time measurements).

In Figure 6 we show an high view of the tool which can lean on the SAClock to compute system precision and support the activity of experimental evaluations. This precision estimator queries the local SAClock, and receives information about accuracy of other nodes. Thus, it computes the system precision and passes this information to the tool which is executing the experimental evaluation.

4.3 Transmission Delay in Quasi-Synchronous Systems

A system is quasi-synchronous ([14], [2]) if: i) it can be defined by the typical synchronism properties: bounded and known processing speed, message delivery delays, local clock rate drift, load patterns and difference among local clocks ii) there is at least one bound where there is a known probability ($\neq 0$) that the bound assumption does not hold (this probability is called assumption uncoverage) iii) any property can be defined in terms of a series of pairs <bound, assumption uncoverage>.

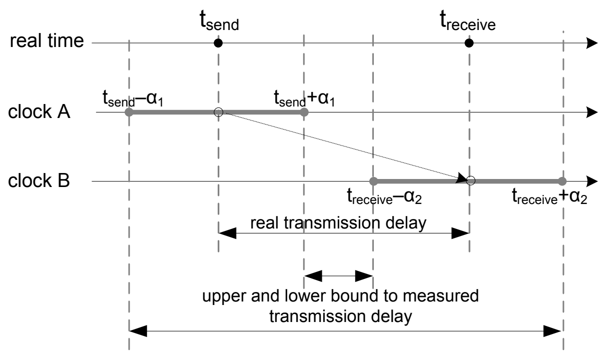

Figure 5. Uncertainty in measuring the one-way transmission delay

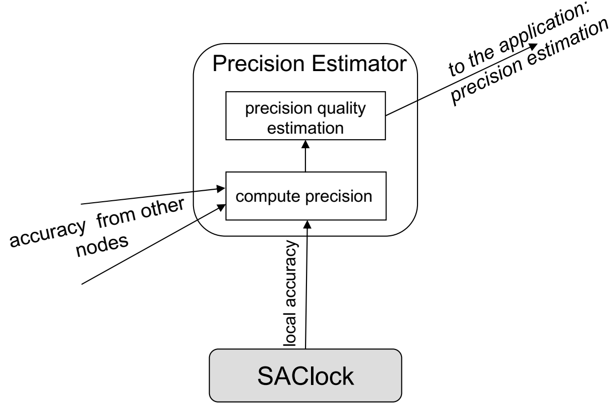
Precision Estimator
precision quality estimation
compute precision
to the application: precision estimation
accuracy from other nodes
local accuracy
SAClock

Figure 6. A precision monitor: a support to tool for evaluation of distributed algorithms

The term quasi-synchronous refers to the fact that these bounds are not precisely known, or the known values are so far from the normal case that in practice different bounds (closer to the normal case) should be used. In such a situation there is an obvious non null probability that the values we pick will not hold.

Let us consider transmission delay. In a classical system only a rightmost bound $max\ T_D$ on transmission delay, with an acceptable residual probability of not holding, would be considered: transmission delay T_D can only be tabulated as accepted (short enough) or rejected (too long). In large-scale pervasive environments this bound would be too large for most applications. In quasi-synchronous distributed system model an application or different applications on the same node may have several bounds for maximum message delivery time, with correspondingly different probabilities of being violated. As example, Figure 7 depicts a situation in which three different bounds T_1, T_2 and $max\ T_D$ of transmission delay are considered, that allow to define four disjoint sectors).

In such a situation, precision influences the decision about the sector in which we place the measured transmission delay. If the precision is high, measurement are placed in each sector with great confidence, while, if precision is low, the uncertainty of the measured transmission delays increases. This uncertainty brings a greater difficult in selecting the sector a transmission delay belongs to.

If the pair of communicating nodes implements the SAClock, they can estimate precision and thus provide a better estimation of uncertainty on measured transmission delays.

4.4 High Dynamic Heterogeneous Environment

Single and entire groups of nodes in a pervasive distributed system may suddenly join or leave the distributed system. They may have different synchronization quality requirements, different synchronization mechanism, and different hardware to achieve clock synchronization (e.g. connections, GPS receivers, even clocks of different quality), applications with different accuracy requirements.

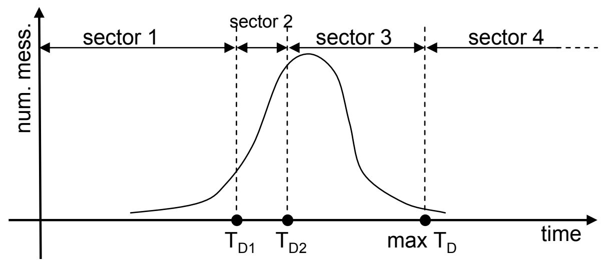

Figure 7. Several applications on a single node may have multiple transmission delay bounds

To accomplish the accuracy and precision requirements of the distributed applications, all these nodes need to communicate each other information about their clocks. The SAClock can help this communication: in fact it allows to exchange and elaborate information on quality of synchronization of i) nodes with different synchronization mechanism (nodes that have switched to another synchronization mechanism are included in this group) ii) nodes with no active synchronization mechanism, thanks to the SAClock caching mechanism (supposed that the synchronization mechanism was previously active).

5 Conclusions and Future Work

In many distributed systems, to achieve clock synchronization is a main task, often not easy to reach. Dynamic networks, faulty nodes or clocks, or even the environment itself may lead to failure in achieving clock synchronization.

This is the reason why applications of pervasive dynamic system, especially those applications that are able to relax their timeliness requirements, can take a great advantage from the presence of a logical clock which, working independently from the synchronization mechanism used, allows the nodes of a network to be aware of the quality of synchronization (node-to-node and node-to-time reference).

In this paper we proposed a new logical clock, which we called SAClock, aware of the synchronization quality of the clock, in term of its accuracy. Applications can take advantage by the use of the SAClock: they can use reliable timestamps, eventually paired to the accuracy of the clocks of other nodes, and they can compute the current precision of the distributed system (precision of the whole system or of a subset of components).

The SAClock is able to make applications continuously aware of the quality of synchronization. This situation can be a great help for elastic/adaptive applications (e.g. real-time mission critical applications and soft real-time applications) that can work using degraded timeliness requirements. We accurately motivated our intention is Section 4, where we proposed several examples of usage.

We are currently developing the SAClock on different platforms and synchronization mechanisms. Moreover, we intend to experiment SAClock on the algorithms presented in Section 4, to measure the reachable improvement in algorithm efficiency using the SAClock and show its applicability.

6 Acknowledgments

This work has been partially supported by the European Community through the IST Project HIDENETS (Contract n. 26979) and by Ansaldo Segnalamento Ferroviario through the Project ATPNS.

References

[1] I. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. Wireless sensor networks: a survey. *Computer Networks*, 38(4):393–422, March 2002.

[2] C. Almeida and P. Veríssimo. Timing failure detection and real-time group communication in quasi-synchronous systems. In *In Proceedings of the 8th Euromicro Workshop on Real-Time Systems*, June 1996.

[3] C. Almeida and P. Veríssimo. Using the timely computing base for dependable qos adaptation. In *20th IEEE Symposium on Reliable Distributed Systems (SRDS'01)*, 2001.

[4] F. Cristian and C. Fetzer. The timed asynchronous distributed system model. *IEEE Trans. Parallel Distrib. Syst.*, 10(6):642–657, 1999.

[5] J. Elson and K. Römer. Wireless sensor networks: a new regime for time synchronization. *SIGCOMM Comput. Commun. Rev.*, 33(1):149–154, 2003.

[6] HIDENETS. Highly dependable ip-based networks and services, annex i - description of work. Technical report, 2005.

[7] H. Kopetz. The time-triggered architecture. *The First IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, page 22, 1998.

[8] J. Lundelius and N. Lynch. A new fault-tolerant algorithm for clock synchronization. In *PODC '84: Proceedings of the third annual ACM symposium on Principles of distributed computing*, pages 75–88, New York, NY, USA, 1984. ACM Press.

[9] D. Mills. Network time protocol (version 3) specification, implementation. 1992.

[10] D. L. Mills. On the accuracy and stablility of clocks synchronized by the network time protocol in the internet system. *SIGCOMM Comput. Commun. Rev.*, 20(1):65–75, 1990.

[11] S. PalChaudhuri, A. K. Saha, and D. B. Johnson. Adaptive clock synchronization in sensor networks. In *IPSN '04: Proceedings of the third international symposium on Information processing in sensor networks*, pages 340–348, New York, NY, USA, 2004. ACM Press.

[12] R. C. R. Meier, B. Hughes and V. Cahill. Towards real-time middleware for applications of vehicular ad hoc networks. Technical report, 1 April 2005.

[13] P. Veríssimo. On the role of time in distributed systems. In *Proceedings of the 6th Workshop on Future Trends of Distributed Computing Systems (FTDCS'97)*, October 1997.

[14] P. Veríssimo and C. Almeida. Quasi-synchronism: A step away from the traditional-fault-tolerant real-time system models. *Bull. Technical Committee on Operating Systems and Application Environments (TCOS)*, 7(4):35–39, 1995.

[15] P. Veríssimo and A. Casimiro. The timely computing base model and architecture. *IEEE Trans. Comput.*, 51(8):916–930, 2002.

[16] P. Veríssimo, A. Casimiro, and C. Fetzer. The timely computing base: Timely actions in the presence of uncertain timeliness. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 533–542, New York City, USA, June 2000. IEEE Computer Society Press.

[17] P. Veríssimo, L. Rodrigues, and A. Casimiro. Cesiumspray: a precise and accurate global time servicefor large-scale systems. *Real-Time Syst.*, 12(3):243–294, 1997.

[18] P. Veríssimo and L. Rodriguez. *Distributed Systems for System Architects*. Kluwer Academic Publisher, 2001.