Sharing a Common Time Reference in a Heterogeneous Distributed System

Augusto Ciuffoletti
Università di Pisa
Corso Italia 40 - 56126 Pisa - ITALY
e.mail: `augusto@di.unipi.it`
URL: `http://www.di.unipi.it/~augusto`

Abstract

This article aims at introducing SYD*, a tool for the generation of time-stamps synchronized with the Universal Time Coordinate: the implementation includes an NTP client and is written in Java*[1]*. Possible applications include (but are not limited to) the measurement of the performance of distributed applications, their debugging, and the implementation of real-time requirements in distributed systems.*

The features that characterize SYD *are its portability across platforms, the low cost in terms of network overhead, and the absence of interference with other applications running on the same host (*SYD *does* **not** *synchronize the system clock!);*

In this paper, we prove that the choice of Java does impact on the performance of the tool, when compared with an equivalent application written in`C`.

SYD *is available on the Web as public domain software.*

1. Introduction

The measurement of time is a key issue in the analysis of the performance and in the control of a system: the analyst may be interested in knowing how much time passes between the occurrence of two events, and the programmer may be interested in scheduling a given time interval between the occurrence of two events.

When the system is composed of distinct processing units, there comes the need of a uniform time reference shared by all cooperating units. This is especially true when the system supports applications that are related to the real time: for instance, a multi-media application may require that simultaneous views of the same phenomenon are synchronized during the playback.

The solution that is often applied in this case (for instance see [4]) consists in synchronizing the *system clock*s of all the system components: this exhibits several drawbacks:

- the periodic re-synchronization of the *system clock* is a very critical activity: abrupt changes in the clock value can damage not only distributed activities, but also the internal consistency of a the host. Many platforms do not provide the way to amortize clock adjustments, and there is no standard way to do this;
- it is not possible to re-synchronize the *system clock*, without suspending and restarting the whole activity of the system component: in fact, a lot of local activities depend on the granularity of the *system clock*. When the adjustment needed to correct the *system clock* is large, a smooth adjustment can be incompatible with the required clock granularity. In case of failure of the clock synchronization tools, the whole system is likely to be restarted;
- the software needed to synchronize the *system clock* runs with extended privileges: therefore it must be installed and maintained by the system administrator. The user has no access to its configuration, and cannot tune it to a specific purposes;
- since each platform implements the *system clock* in a different way, the applications that use it inherit some form of non-portability: this contrasts with the need of software portability, even across platforms.

[1] Java is a trademark of Sun Microsystems, Inc.

- also the clock synchronization software suffers from the diversity of the *system clock* implementation it must cope with: the *xntpd* project is an example of the effort needed to support a clock synchronization daemon portable to almost any Unix operating system.

Although there are significant cases where the above drawbacks must be accepted and coped with, since the *system clock* synchronization is the only design opportunity to solve some problems (for instance, in a control system with hard real-time requirements), in a lot of cases it is the kind of solution that over-kills the problem introducing unnecessary reliability problems and design problems.

In most cases, the need for a universal time coordinate is limited to few activities, and it is not needed to (or even must not) influence the scheduling rate, or other activities that depend on the *system clock*.

When the problem to solve is measuring the performance of a distributed system, or setting a common deadline for a certain activity, the programmer needs the answers to questions like:

- how long this message has been delayed by the network?
- is the deadline for the activity already expired?
- which event, in a given set, took place first?

To answer all these questions, every system component needs to be able to produce time-stamps that reflect the present value of a universal time coordinate [3]. This is a much more limited objective than keeping the *system clock* of the system components permanently synchronized.

In this paper we are presenting a tool that is able to produce accurate time-stamps that reflect the Universal Time Coordinate (UTC). The tool consists of a process that is running permanently (a *daemon* in the Unix terminology), and that periodically retrieves the time value from a set of NTP servers. The NTP (Network Time Protocol) service [4] is an Internet service based on a standard protocol that diffuses accurate timestamps through several servers. NTP servers are available worldwide, and a directory can be found at `http://www.eecis.udel.edu/~mills/ntp`.
Based on the timing information retrieved from these accurate sources, the daemon computes and keeps updated a small data structure, which is called *virtual clock* from [2], that maintains the data needed to correct the value of the internal *system clock* in order to obtain time-stamps that reflect the value of the UTC standard with a bound approximation. We can conclude that the virtual clocks (i.e., the time values that can be obtained by correcting the *system clock* value with the content of the virtual clock) are synchronized.

The frequency with which the *virtual clock* is adjusted, asking a new time value to the NTP server, is quite low, being of the order of the minutes, and it heavily depends on the required approximation. Therefore the network overhead is regarded as negligible.

The SYnchronization Daemon (of which *SYD* is an acronym) acts as a client with respect to the remote NTP server, and has also a server interface for the local activities that need to obtain accurate time-stamps. The service provided by *SYD* does not consist in passing the time-stamp itself, but the virtual clock, the data structure needed to correct the *system clock* value. In order to compute a time-stamp, the client will apply the appropriate function, chosen from a set contained in a library which is shipped together with the *SYD* code, to the data structure (whose validity can be periodically checked). Each time-stamp has attached an indication of its precision, so that the client can choose to periodically *refresh* the local *virtual clock* structure.

The architecture introduced so far eliminates the drawbacks mentioned above: the operation of *SYD* is as critical as the activities that explicitly depend on it, and does not impact on activities that do not need an accurate time-stamping. A *SYD* daemon can be started or restarted whenever necessary with no impact on the system component, except for the activities that explicitly depend on its activity. Although unnecessary, several instances of *SYD* can be run concurrently on the same system. And, finally, *SYD* does not need any special privileges in order to be run, since it does not need any critical resource.

A final point is about portability: having a software component that freely migrates from one platform to another means that the analyst or the programmer needs to learn only how to configure and use the tool, independently from where it is located.

In order to reach this further goal we opted for writing *SYD* using Java. The final result is an implementation which migrates with zero porting effort from Windows95 to Linux.

2. The structure of *SYD* a SYnchronization Daemon

The cornerstone of the architecture of *SYD* is the **Daemon**: in essence it maintains a **VirtualClock** object by interacting with several NTP servers, and passes such information to the applications that ask for it.

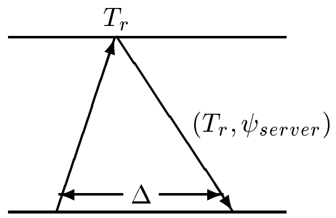

Figure 1. The interaction with the remote NTP server implemented by the `read` method

The internal structure of *SYD* can be described as the composition of a number of simple modules:

the remote clock reading rule that is in charge of computing an approximation of the present UTC time;

the virtual clock maintenance algorithm that generates and keeps up to date the virtual clock;

the selection algorithm that checks different virtual clocks for consistency, in order to cope with server failures;

the responder , a separate *thread* that delivers the virtual clock data structure to the clients that ask for it.

According to the object oriented nature of Java each functionality has been encapsulated into an object.

2.1. The remote clock reading rule.

The remote clock reading rule consists of two steps:

- an exchange of data between *SYD* and the remote NTP server and
- the computation of a 3-ple (t_l, t_s, ψ) with the meaning: "when the local *system clock* shows t_l the NTP server clock shows a time within $[t_s - \psi, t_s + \psi]$", using the data gathered in the first step.

Concerning the first step, we have created a class **RemoteClock** whose role is performing the message exchanges with the remote NTP server. The data remain, after the data exchange, as a part of the state of the object.

The `read` method of the **RemoteClock** measures the delay between a request of connection to the NTP socket on the remote server, and the receipt of the reply (see Fig. 1), containing a timestamp T_r corresponding to the time when the reply was sent, and the estimated accuracy ψ_{server} of the server's clock.

The second step is carried out by the `retry` method of the objects from the class **Session**. The operators used to compute the three values are illustrated in [1], and their exhaustive justification is outside the scope of this paper.

Basically, the information gathered during the previous data exchange is combined with other local data, such as the minimum communication delay *min* and the local clock accuracy, in order to compute the triple (t_l, t_s, ψ), that will be recorded in the state of the object. The detailed description of the rule to compute these values is in [1]. Here we only introduce an upper bound of the accuracy ψ:

$$\psi \leq \frac{\Delta}{2} - min + \psi_{server} \tag{1}$$

that will be used to evaluate the performance of the programs, in the last section.

2.2. The virtual clock maintenance algorithm.

This functionality is encapsulated into the **Session** object.

The `attempt` method repeats as many `retry` methods as needed to obtain a precision ψ lower than a given threshold ε. In fact, *SYD* is bound to keep the accuracy below a certain threshold. According with a probabilistic model of network delays (that the user can derive from the log produced by *SYD* itself) this threshold may not be reached in a single retry. Given the probability $p_r(\text{fail})$ of failing to reach the required accuracy in a single retry, and the number k of retries available for an attempt, we can compute the probability $p_a(\text{fail})$ that the attempt fails to reach the required accuracy:

$$p_a(\text{fail}) = (p_r(\text{fail}))^k \tag{2}$$

In this case the attempt terminates throwing an exception.

As soon as $\psi \leq \varepsilon$, the `attempt` method returns. From that time, the *virtual clock* starts drifting away from the UTC clock at the same rate as the *system clock*. Thus, if we want to keep the *virtual clock* accuracy below a given value, we have to schedule another `attempt` before a certain timeout expires. This task pertains to a *thread* implemented by **Sessions** class. The periodical invocation of an `attempt` pertains to the **Sessions** class, that implements a *thread* whose task is keeping an object of the **VirtualClock** class up to date.

Table 1. Command line options

Option	Units	Description
`-h`	address	the remote host address
`-m`	msecs	min
`-e`	msecs	ε
`-d`	msecs	δ
`-w`	msecs	W
`-k`	n	k
`-r`	ppm	ρ

The interval between two invocations depends on the drift estimate ρ of the local *system clock*, and on the upper threshold for the accuracy δ. The timeout for the next *successful* retry can be expressed as:

$$\frac{\delta - \varepsilon}{\rho}$$

Since we must take into account that at most k attempt might be necessary before a *successful* one, each separated by a lapse W to ensure the independence between the attempts, the following inequality must hold:

$$\frac{\delta - \varepsilon}{\rho} - k * W \geq 0 \tag{3}$$

2.3. The selection algorithm.

As noted above there is a non null probability that a **Session** fails in absence of external failures. But this probability is usually negligible with respect to the possibility that the **Session** fails because of a temporary shutdown of the NTP server it is connected to, or for some other unexpected events. To cope with such exceptional events, the **Daemon**, which contains the `main` method of *SYD*, activates several **Session** *threads*, each attached to a different NTP server.

The principal session is configured according to the parameters passed as command line parameters, in a Unix-like way: Table 1 lists the option name followed by the units and the meaning of the option parameter.

Other sessions are described in a configuration file `syd.conf`: each line contains the Internet address of the remote host, and the values corresponding to the variables min, ε, δ, W and k suitable for that host. Each session has a specific log file, that contains a human readable, detailed description of each retry. Log files are stored, together with the `syd.conf`, in a directory named `log_files`.

Each **Session** *thread*, in its turn, informs the **Daemon** about two kinds of events: successful refresh of the **VirtualClock** attached to the **Session**, or failure to maintain the required accuracy.

Based on these notifications, and on the internal states of the **VirtualClocks** attached to the **Sessions**, the **Daemon** can select at each time the virtual clock that is better suited to be delivered to the clients. Such indication is used by the *thread* described in the following section.

The class **Daemon** logs its activity in a separate log file, `log_d`, and its `main` method is invoked with the command `java syd.Daemon` followed by the options indicated above.

2.4. The responder

This activity consists in responding to the requests from the clients, and is carried out by a separate *thread*, activated by a **Daemon**. It implements the server interface of *SYD*.

One interesting option for this interaction between the client and *SYD* has been the use of the *Remote Method Invocation* available for Java. We finally opted for a much less sophisticated (but more complex to implement) interaction through a socket. This has two important advantages:

interoperability – by using the *Remote Method Invocation*, the service offered by *SYD* is limited to applications written in Java, while the interaction through sockets is available to any language for which a socket library is provided;

speed – a *Remote Method Invocation* entails a complex protocol between the server and the client when the client applies a method to the remote object: this can be time consuming, while the operations to be applied are time critical. If only the state of the object is passed to the client, and the client imports the class, we obtain a much more predictable timing of the operations.

However, the option we have chosen requires that the data passed through the channel are explicitly marshalled on the server site, and unmarshalled at the client side: but this is a minor drawback compared with the above mentioned advantages.

3. The VirtualClock

This is the class of the objects passed by *SYD* to its clients. Its implementation is of some interest, since other applications written in Java can use the methods of this class to manipulate the object received from *SYD*. Applications written in other languages will have to build an appropriate library on their own in order to perform the operations needed to use the data obtained from *SYD*.

unmarshalling – this is the basic operation that reconstructs the **VirtualClock** starting from the byte stream received through the socket which is opened with *SYD*. The result is a new object of class **VirtualClock**.

test – there are cases where the **VirtualClock** obtained from *SYD* is not valid (yet). This method returns this property of the state.

reading – which consists in reading the clock. There are two basic figures that are interesting: the *value* and the *precision* of the clock, and two distinct methods return them, whose operation is straightforward. The former is obtained by adding the offset of the *system clock*, with respect to the UTC time (computed by *SYD* and recorded into the state of the virtual clock), to the current value of the *system clock*. The latter is the time elapsed from the last adjustment of the **VirtualClock**, times the maximum estimated drift of the *system clock*. Both figures can be either referenced to the present time, or to another time, in the past or in the future. This is an option that is valuable when writing time critical software.

The interested reader may find in the appendix the synopsis of the methods exported by the **VirtualClock**, and a small example illustrating the use of *SYD* in a toy application.

4. Results

Once the prototype version had been up and running, we investigated some key issues about some design options.

4.1. Is Java appropriate for this task?

The basic performance parameter of a time-stamp generator tool is the accuracy of the time-stamps it produces. Since such value depends on the timing of the interaction with the server, and deteriorates if the timing is loose, the adoption of Java seems questionable. In fact the partially interpreted execution model of Java might impact on the network interface overhead.

Therefore we have decided to compare the performance of the Java implementation with the one of a prototype version of *SYD* written in `C`, that we expect to be faster. Both applications were run on the same machine, a 90 Mhz Pentium with a Linux Operating System.

round-trip delay (msecs)		
Language	minimum	median
C	47	71
JAVA	48	72

Table 2. Minimum and median of the round-trip delay

In figure 2 we have summarized the distribution of the round-trip delays measured by the program written in `C`, and in figure 3 there is a similar diagram for Java. The two distributions are very similar, and refer to two simultaneous experiments that lasted approximately 2 months, gathering more than 2000 round-trip delays. In table 2 we have summarized two basic statistical parameters from the experiments: the minimum round-trip delay, and the median of the round-trips.

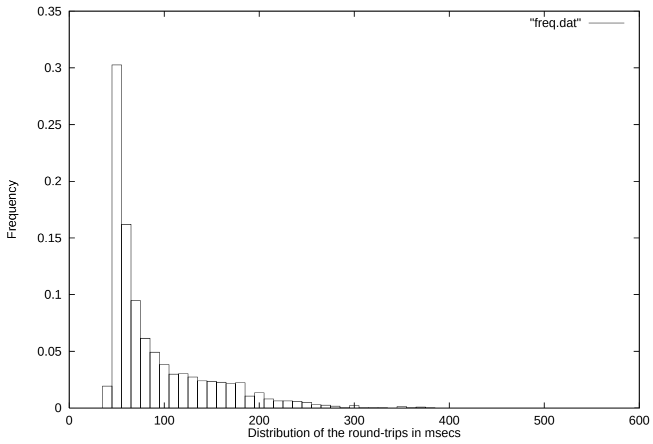

Figure 2. Distribution of the round-trips for C

In order to quantify more precisely the performance of the two programs, we derived an analytical expression for the best accuracy reachable under certain conditions.

The inequality (3) can be rewritten as:

$$\delta \geq \rho W k + \varepsilon \stackrel{\text{def}}{=} d_{min} \tag{4}$$

where we introduce d_{min} as the lower bound for the accuracy.

The data summarized in Fig(s). 2 and 3 are used to compute the probability of failure of a single retry $p_r(\text{fail})$, since, for the (1),

$$p_r(\text{fail}) = p(\Delta \geq 2(\varepsilon + \min - \psi_{server})) \tag{5}$$

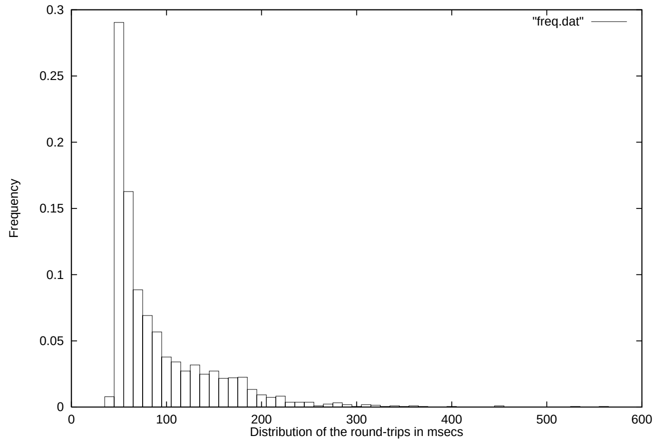

Figure 3. Distribution of the round-trips for Java

From the experimental data available for a given setup, we can infer the inverse function $\Delta(p)$ defined as follows:

$$\Delta(p_0) = \Delta_0 \text{ such that } p(\Delta \geq \Delta_0) = p_0 \quad (6)$$

From this definition, and the (5)

$$\varepsilon = \frac{\Delta(p_r(\text{fail})}{2} - (\min + \psi_{server})$$

Combining it with the definition of $p_a(\text{fail})$ in equation (2) and the definition of d_{min} in equation (4), we obtain:

$$d_{min} = \rho W k + \frac{\Delta(\sqrt[k]{p_a(\text{fail})})}{2} - (\min + \psi_{server}) \quad (7)$$

This is a function that binds the best accuracy with a number of parameters: while many of them can be considered as dependent on the environment, the k, the maximum number of retries, appears as a design option.

We recall the meaning of the other parameters, and give a value that we will consider as "typical" for that parameter:

$p_a(\text{fail})$ the desired reliability of the program (typ. 0.01);

ρ the maximum drift of the *system clock* (typ. 0.0001);

W the lapse between successive retries that guarantees that the delays are independent (typ. 60 seconds);

ψ_{server} the worst case accuracy of the server (typ. 5 msecs);

Table 3. Optimal setup for the typical scenario

Language	k_{opt}	δ_{opt} (msecs)	ε_{opt} (msecs)
C	5	53.2	23.2
Java	5	54	24

$\Delta(p)$ a function that, given a probability p, returns the value Δ_0 such that $p(\Delta \geq \Delta_0) = p$. This function can be obtained by the samples illustrated in Fig(s). 2 and 3.

The value of min can be obtained as $\frac{\Delta(1)}{2}$.

The experience justifies the above mentioned "typical" values as slightly pessimistic, but otherwise reasonable.

Once such values are defined, the (4) is a function of the number of retries, k. This function is characterized by a minimum, that corresponds to the optimal value of the guaranteed accuracy: we denote with δ_{opt} and k_{opt} the values that characterize the saddle. From inequality (3) we can directly compute the corresponding value for ε, that we denote with ε_{opt}.

We choose the value δ_{opt} as a metric for the performance of the algorithms. The values, computed for a "typical" scenario and for the roudtrip delay distributions illustrated in Fig(s). 2 and 3, are summarized in Table 3.

We conclude that under the "typical" conditions the program written in `C` could maintain an accuracy of 53.2 msecs, while Java could maintain an accuracy of 54 msecs. The difference between the two is negligible.

4.2. Is the probabilistic approach worth?

A probabilistic approach involves some additional effort both in the design of the client of the synchronization service, and in its configuration. The alternative is a deterministic protocol, that starts from the assumption that roundtrip delays do not exceed a given threshold. This corresponds to a probabilistic solution that allows a single attempt, with a high probability of success.

Our results prove that the option of repeating several attempts with a low probability of success allows a higher accuracy with respect to a single attempt with a high probability: in fact the saddle point corresponds to a set-up that envisions 5 retries, each with a probability of success of 60%. A sub-optimal set-up allowing just one attempt (with a probability of success of 99%), could reach an accuracy δ_{min} of 215 msecs in Java, and

Table 4. Frequencies of the number of retries before success (n_r) on a sample of 100 attempts

Frequencies of n_r					
n_r	1	2	3	4	5
Expected	0.60	0.24	0.10	0.04	0.01
Results	0.83	0.16	0.00	0.00	0.01

127 msecs in C, making the difference between the two implementations dramatic!

4.3. Does a probabilistic approach return the expected results?

The probabilistic approach derives the design from a probabilistic model of the system. The test "on the field" verifies the fitness of the design to the real world.

Table 4 shows the expected and the real frequencies of the number of retries before success (n_r) for the optimal setup indicated above for the program written in Java, during an experiment which had been lasting for 2 days (2000 attempts, at the rate of one every a few minutes): the design is slightly pessimistic, but sufficiently accurate.

4.4. Are successive attempts really independent?

One of the most critical assumptions is that successive retries are independent.

Although this is in fact a special case of the last point, we made a specific experiment to assess this, measuring round-trip delays separated by less than 6 seconds, an lapse 10 times shorter than the typical W.

Figure 4 summarizes the results: each point corresponds to a couple (Δ_{n-1}, Δ_n) of successive round-trip delays. In that representation, the dependency between successive round-trips would be indicated by the aggregation of the points around the diagonal line $x = y$: in our case, there is not such evidence.

4.5. What is the maximum attainable precision of this implementation?

The bottleneck is still in the network link layers, and below: we have reached the maximum precision allowed by the Java language clock granularity (1 msec) operating in loop-back, thus virtually with no network delay. The C implementation can reach accuracies below such limit.

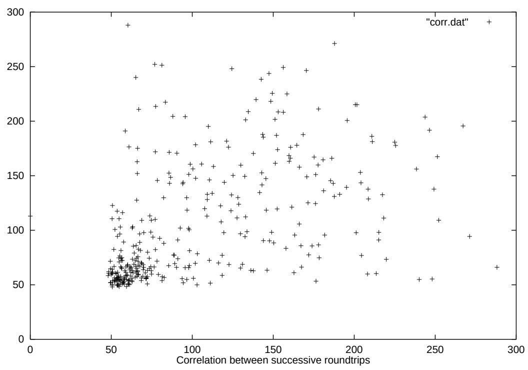

Figure 4. Correlation between successive round-trip delays

5. Conclusions

The results shown above confirm that the choice of Java for the implementation of a probabilistic network clock synchronization algorithm is realistic, and does not exhibit any apparent drawback when compared with another implementation written in C.

SYD is currently available as public domain software following the GPL policy. The reader is invited to down-load[2], try, and use it. The package contains the tools to analyze the log files generated by *SYD* and to obtain the figures that illustrate this paper.

6. Acknowledgments

The activity described in this paper has been carried out during a visit at the International Computer Science Institute in Berkeley (CA). Here I wish to extend my thanks to Andres Albanese, leader of the "Network" group, and to Giancarlo Fortino, that helped me in the first steps with Java.

References

[1] G. Alari and A. Ciuffoletti. Implementing a probabilistic clock synchronization algorithm. *Real-Time Systems*, 13(1):25–46, July 1997.

[2] F.Schmuck and F.Cristian. Continuous clock amortization need not affect the precision of a clock synchronization algorithm. In *9th Annual ACM Symposium on Principles of Distributed Computing*, pages 133–143, 1990.

[2]The software can be found at `http://www.di.unipi.it/~augusto/` and following the link "Syd")

[3] L. Lamport. Using time instead of timeout for fault-tolerant distributed systems. *ACM Transactions of Programming Languages and Systems*, 6(2):254–280, April 1984.
[4] D. Mills. Internet time synchronization: the network time protocol. *IEEE Transactions on Communications*, COM-39(10):1482–1493, October 1991.

A. Virtual clock methods

A.1. Constructor Index

VirtualClock (byte[]) CONSTRUCTOR: creates a new virtual clock using an array of bytes to determine the relevant fields.

VirtualClock (double) CONSTRUCTOR: creates a new dummy virtual clock, specifying only the maximum drift of the system clock.

A.2. Method Index

commit() MUTATOR: Commits this clock as accurate.

is_committed() ACCESSOR: Tests whether this clock is accurate or not.

is_valid() ACCESSOR: Tests whether this clock can be used or not.

offset() ACCESSOR: Returns the offset of the virtual clock with respect to the system clock.

precision() ACCESSOR: Returns the current precision of this virtual clock.

precision(long) ACCESSOR: Returns the precision of this virtual clock at a given time.

set(long, long) MUTATOR: Sets the current value and precision of this virtual clock.

set(long, long, long) MUTATOR: Sets the value and precision of this virtual clock at a given time.

toBytes() ACCESSOR: Encodes the relevant variables into a new byte array.

toString() ACCESSOR: Translates the value of this virtual clock into a human readable string.

value() ACCESSOR: Returns the current value of this virtual clock.

value(long) ACCESSOR: Returns the value of this virtual clock at a given time.

B. A toy example

The following application shows the basic steps to use a virtual clock provided by *SYD*, once the syd **Daemon** is running:

- create a socket to the *SYD* **Responder**
- download from that source the byte stream containing the description of the **VirtualClock**;
- generate a local instance of the **VirtualClock** with the state downloaded from syd;
- use the value of the time, as indicated by the virtual clock. In this case the value of the virtual clock is displayed every second.

```
package syd;

import java.net.*;
import java.lang.*;
import syd.virtualclock.*;

public class Example {
  public static void main(String args[])
    throws Exception {
    byte[] send_buffer = new byte[48];
    InetAddress address;
    DatagramPacket packet;
    DatagramSocket socket;
    byte[] buffer;
    VirtualClock vc;

    // creating socket
    socket = new DatagramSocket();
    // sending packet through socket
    address = InetAddress.getByName("local...
    buffer = new byte[40];
    packet=new DatagramPacket(buffer,40,ad...
    socket.send(packet);
    // set the timeout
    socket.setSoTimeout(1000);
    packet=new DatagramPacket(buffer,40);
    // waiting for response
    socket.receive(packet);
    // constructing new clock
    vc = new VirtualClock(buffer);
    while (true) {
      System.out.println(vc.toString());
      Tread.sleep(1000);
    }
  }
}
```