GaliLEO: a simulation tool for LEO satellite constellations [*]

Laurent Franck
ENST/TéSA
Toulouse (FR)
Laurent.Franck@enst.fr

Francesco Potortì
CNUCE-CNR
Pisa (IT)
F.Potorti@cnuce.cnr.it

Abstract

We present Galileo, a simulator for the transmission of both connection-oriented and connectionless traffic over a constellation of LEO/MEO (Low / Medium Earth Orbit) satellites. Its scope is limited to the satellites and the stations accessing them, without any modelling of the terrestrial network, but inside this scope the goal is to study the performance of satellite-based communication networks from as many as possible points of view. Typical applications include simulation of access techniques, routing policies, fault management. The simulator is written in Java, and it makes use of dynamic loading to easily integrate user-written modules. A draft manual is available, and a preliminary version of the program will be published by the end of 2000.

1 Introduction: the newborn and its family

The motivation behind Galileo's conception emerged during an exchange of ideas among some members of the European COST 253 Action[1], a forum where researchers from all around Europe periodically meet to address issues related to LEO constellations of communication satellites. The point made was that none of the commercially or freely available simulation tools was reasonably usable as a generic simulation tool for LEOs. This seminal discussion later led to the initial design of the Galileo architecture, which was the outcome of a collaboration between two institutes where researchers had already had experiences in developing special purpose simulators.

The basic idea and most of the concepts regarding the connection setup and the channel access from a ground station were developed at CNUCE, an institute of CNR in Pisa (IT), as a consequence of the inadequacy of the locally developed Fracas [**?**] simulator for the study of LEO networks. The routing concepts and the details of the architecture that form the glue of the actual implementation of Galileo come from the experience of the LeoSim [**?**] simulator, developed originally at the Brussels University (BE), and currently at ENST in Toulouse (FR), and the initial specifications of SimToc [**?**], designed at CNUCE. Interestingly, while having very different scopes and objectives, all of these simulators took an object oriented ap-

[*]This work was partially supported by the CNR (Italian National Research Council) under the 5% Multimedia Programme.

[1]COST stands for *COoperation in the field of Scientific and Technical research,* see <URL:http://www.eeng.brad.ac.uk/Research/cost253/> for information on COST 253.

proach to implementation, principally as a mean to ease extensibility. Galileo aims to be a general purpose, customisable tool, freely available for the whole satellite community.

Fracas (FRAmed Channel Access Simulator) is essentially a command line driven emulator whose time advances in fixed length steps, usually of the same length of a frame of the protocol under study. While very fast and very well suited to the study of access protocols for GEO systems, it cannot be adapted to LEO systems. Its heritage consists of the concepts behind statistics collection and manipulation.

LeoSim, the most important of Galileo's ancestors, is an event-driven, continuous time simulator accessed through a graphical interface. It has been developed at ENST (FR) in order to study link state routing algorithms for LEO satellite constellations. LeoSim provides statistics on the number of call requests, the call block probability, and the cost introduced by maintaining the link state database; shortest path routing, handover management and elaborate routing signalling are implemented. Its design approach and its core simulation engine have been transported into Galileo.

From SimToc, the other ancestor from CNUCE, Galileo took the global architecture, the idea of the up-down link between the ground station and a satellite of the constellation, and the way a connection is set up and modified. SimToc has never gone past the design stage.

Consim [**?**] is a simulator developed at CSELT (IT) for evaluating the performances of constellations of communication satellites affected by different types of failures. Consim will be integrated by results into Galileo. By "integration by results" we mean that the two simulators are kept separate, and the results of Consim are used by Galileo. This is a simple way to program interaction between two simulators that were written separately, while minimising the coupling needed between the different teams responsible for the programs. However, this approach is only feasible when the two studies cover aspects which are not interdependent. In this case, Consim runs a failure model to produce a list of failure events occurring during the constellation lifetime, each one tagged with the type of failure and the time of occurrence. Since the fault occurrence is independent of the traffic generation, Consim needs no feedback from Galileo, and the data exchange between the simulators can be unidirectional. In practice, the list of failure events is provided during the simulation initialisation phase and then used to feed Galileo's simulator engine in order to trigger the right fault managers at the appropriate time.

2 Architecture

Figure 1 highlights the three-layer architecture of Galileo.

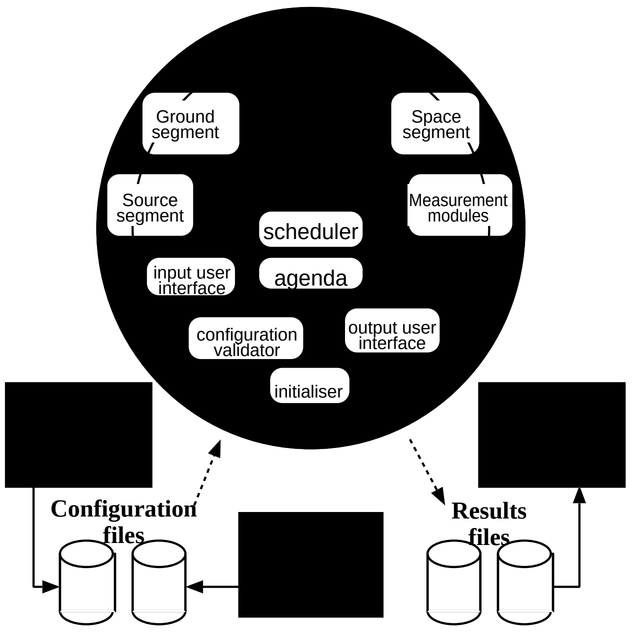

Figure 1. Galileo architecture.

The first layer (*Simulation engine*) defines both the structure of *components*, which are the blocks from which Galileo is built, and the way they behave and interact. The components' dynamics is the job of the scheduler that runs the components and defines a message passing structure for inter-component communication. Thus the first layer constitutes a generic infrastructure for a discrete-event, message-passing simulator.

The second layer (*Core modules*) implements the *network model* of Galileo, by defining the scope of Galileo possibilities through the definition of a set of classes. These classes (called *templates*[2]) also specify the rules for using the network model and creating custom components. This layer is not customisable per se, and in fact is the core of Galileo's functionalities. Experimenters needing to implement their own set of purposefully made modules should be well acquainted with the network model.

The third layer (*Custom modules*) is the set of modules which are dynamically loaded from a library, including both *standard* and *custom components*. Standard components are components shipped with Galileo. Custom components are developed (possibly based on standard components) by the user of Galileo in order to tailor the simulator to its needs. This layer is where ad-hoc built modules are integrated in Galileo. Examples include modules defining the behaviour of actual routing algorithms, channel allocation methods, traffic generators, call admission control policies, etcetera.

To summarise the relation between the second and third layer, layer two defines what are the general characteristics of, say, a channel allocation method (in terms of what are the provided services) while layer three defines actual channel allocation methods.

2.1 Components as building blocks

Galileo is extremely modular, because it aims at providing a simulation framework where one plugs in a locally developed, e.g., routing algorithm, and evaluates the resulting behaviour. The basic module is called a component, which is a class of Java objects that provide the ability to be duplicated, and methods to initialise and start themselves after creation. *Initialisation* may be based on the presence of other components in the system, and may make use of a dedicated section in the initialisation file, whose sections can be structured and nested to arbitrary depths. *Starting* a component is done after initialisation. This usually makes sense only for *entities* (modules with a special processing capability), which are described below. Galileo comes with a small collection of standard components, which are meant to be used as-is or replaced with custom ones. Hopefully, Galileo's library of standard components will grow with time.

Standard and custom components are built upon templates, which are Java abstract classes used to provide an API for the development of components. Providing an API has some shortcomings with respect to providing an extension language; for example, it is generally more difficult to program in Java than in an extension language, an extension language can be limited to provide only special constructs, and can be well insulated from the details of the program core, which in practice is impossible in Java. However, using an API is a far easier and more flexible approach, and certainly more efficient in terms of resources usage.

Components contain both code and data. After initialisation, the component can either live as a passive element, whose methods are called by the system or from other components, or behave as an independent piece of code. This latter special kind of component is called an entity. Entities run concurrently with the rest of the system and other entities, by using the communication and scheduling facilities provided by the simulation engine. The Galileo network model described later is therefore a collection of interoperating components.

[2] They are unrelated to C++ templates.

3 The simulation engine

The simulation engine comes from LeoSim, and includes the scheduler and the agenda.

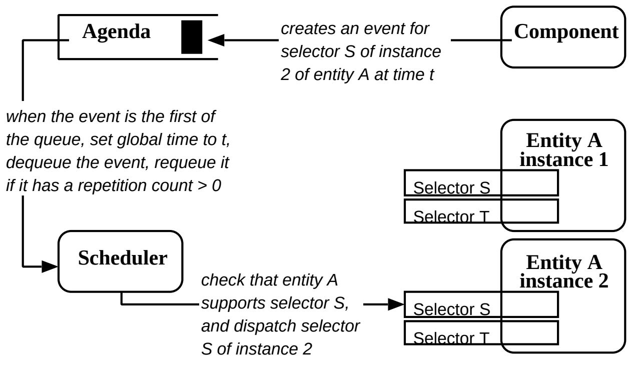
Agenda
creates an event for selector S of instance 2 of entity A at time t
Component
when the event is the first of the queue, set global time to t, dequeue the event, requeue it if it has a repetition count > 0
Entity A instance 1
Selector S
Selector T
Scheduler
Entity A instance 2
check that entity A supports selector S, and dispatch selector S of instance 2
Selector S
Selector T

Figure 2. Creating, scheduling, and consuming events.

Any module inside Galileo can generate an event by calling the `scheduleAction` method of the *scheduler*, which creates a pending event. The arguments of this method are the delay after which the event should be triggered, a repeat count, and the action triggered by the event. The scheduler organises the pending events in a structure called an *agenda*, which is conceptually a queue where the events are kept sorted according to the time when they should be triggered. The exact implementation of the agenda is customisable, to allow experimentation, easy upgrading, and platform-specific optimisations. Currently, a simple-minded delta list is implemented, together with a more sophisticated calendar queue implementation.

A *delta list* is a structure allowing basically two operations, namely insertion of a random element and extraction of the smallest element. The implementation consists of a linked list where an event is inserted in order, so the extraction consists simply of extracting the first element of the list. Unfortunately, inserting an element needs scanning the list from the beginning, so the insertion time is $O(N)$, N being the number of pending events. For big simulations, like those we are planning, this is not acceptable, because the overhead introduced by event scheduling would be $O(N^2)$. A *calendar queue* is a structure allowing the same operations as those provided by a delta list, but using a more complex data structure, consisting of an array of linked lists. It is possible to make an analogy with a calendar, where for each day one writes down zero or more appointments, ordered by the time of the day. Finding the day where the appointment should be written is $O(1)$, and inserting it in the queue of that day is $O(n)$, n being the number of events (appointments for that particular day). Two parameters must be set for a calendar queue, that is the slot size, which is the length of the day in the calendar analogy, and the number of slots. It has been shown in [**?**] that the optimal number of slots is $O(N)$. By changing dynamically the number and the size of the slots depending on N we obtain a *dynamic* calendar queue, for which empirical evidence has been given in [**?**] that insertion and extraction times are $O(1)$. Currently, Galileo implements a *static* calendar queue, whose parameters are read from the configuration.

The action triggered by an event is defined by a *selector* and a list of arguments to it. A selector is an entry point in a module, that is, a method which possibly accepts arguments. When an event is triggered, the associated selector is called, and the relative list of arguments is passed to it. This simple message passing mechanism allows asynchronous communication between components. More precisely, any piece of code inside Galileo can generate an event, and thus send a message, but only entities can have selectors, and thus be awakened by the scheduler and receive a message.

The scheduler is the heart of the simulator. After the initialisation phase, the simulation consists of a loop running inside the scheduler, which just removes the first event from the queue, advances the system's time to that of the event, and calls the selector specified therein, with the appropriate argument list. When the selector is finished, it returns to the scheduler. The loop finishes when there are no more events in the queue, a special stopping event is encountered, or when manually stopped by the operator.

Since it is anticipated that Galileo will go distributed in the future, the scheduler is customisable, to allow experimentation and local customisations of distributed scheduling criteria. Currently, a simple serial scheduler is available, which is the normally used one, and a parallel scheduler is implemented, which is useful for multiprocessor machines.

After selecting a first event for running on a given CPU, the scheduler could remove a second event from the head of the queue to have it run on a second CPU. This is always possible if the selector has the same trigger time as the first one, and the Java code is written with parallelism in mind (i.e., by properly using the `Synchronized` statement or modifier). In the general case, however, the second selector should be run only if the first selector does not change any state in the simulator on which the second selector depends. This constraint cannot be automatically detected, so selectors wishing to allow parallel execution of other selectors must provide an `isSafeWith` method, which takes the next selector as an argument. In our example, the first selector's `isSafeWith` method would be called with the second selector as an argument, and should return `true` only if the second selector is known not to rely on any state that the first selector may change.

Moreover, if the first selector creates any events whose time is less than the time of the second event, these events (recursively) should not trigger any change of state on which the second selector relies. In order to ease this requirement, a second argument is passed to the `isSafeWith` method, which is the time of the second event.

This mechanism is far from being automatic, but in practice it can allow some parallelism for carefully crafted components that have been written by the same programmer. For example, computing a route is a time consuming task (it involves usually a shortest path algorithm). If two successive routing events are present in the agenda, they could be launched concurrently provided that they do not have to be performed in the same satellite.

4 The network model

The second layer of the architecture depicted in Figure 1 defines the basic capabilities of the simulator as far as the modelling of the communication network is concerned. The relevant modules are the Source, Ground and Space segments. Each is a collection of components and templates. Custom and standard components are instantiation of templates, and occupy the third layer of the architecture. Galileo will initially ship with a small set of standard components, and a manual describing the API for building custom ones.

4.1 Assumptions and definitions

Many components in Galileo are meant to describe real objects in the satellite network. We describe the main concepts used when describing the network, and when there is a direct correspondence between a concept and a component, we will indicate the name of the component in monospaced face between brackets, like in [`Satellite`].

We define a *cell* as the area of the earth illuminated by a satellite spot beam. A *footprint* is the whole coverage area of a satellite [`Satellite`], i.e. it is the sum of the areas covered by its spot beams. An *overlap area* is the area in which a *ground station* [`Station`] (i.e. a single subscriber or a concentrator) can receive a signal with an acceptable power level from more than one adjacent spot beams. A *UDL* (Up-Down link) [`Udl`] is the aggregation of all spot beams pertaining to the same footprint; it has a fixed capacity, and is unidirectional. A *beam* [`Beam`] is the

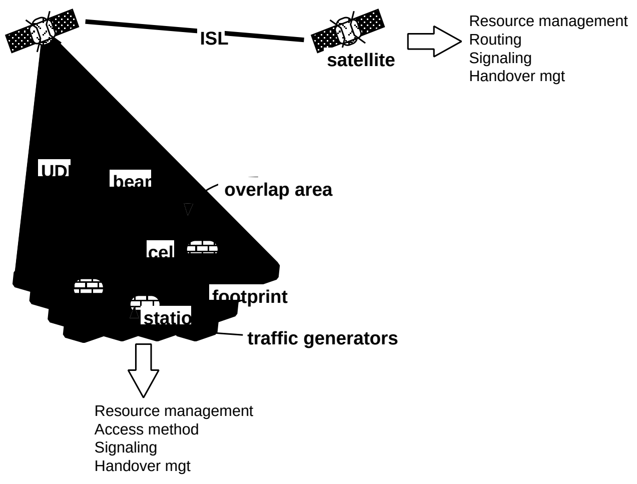

Figure 3. Names of some objects used in the simulator.

communication medium between a satellite and a spot on the ground. A beam has a variable capacity which cannot exceed the capacity of the UDL the beam belongs to. A *node* [`Node`] of the network is any station or any satellite. Satellites have multi-beam antennas for up-link reception and down-link transmission, and are connected to each neighbouring satellite by means of an *ISL* (inter-satellite link) [`Isl`] which a unidirectional link.

A *connection* [`Connection`] is a virtual communication path between a *source* and a *destination*, which are normally different stations. A connection can be created, modified by changing its characteristics [`Resources`], and torn down. It is assumed to be full-duplex, composed by a forward and a return channel [`UniConnection`], where the forward channel is intended to be from source to destination, and the return channel from destination to source.

The procedure supporting the transition from one connection state to another is implemented by a call signalling protocol [`CallSignaling`].

4.2 Call signalling

A *call generator* [`CallGenerator`] defines when a connection starts, between which endpoints, and how and when it is modified and torn down. It can be associated with a *packet generator*, which produces the packet traffic running over a connection, for simulating connection-oriented traffic. It is envisaged that Galileo will be able to support also traffic generators that create connectionless traffic. In the following we will mainly consider connections and connection-oriented traffic.

When a connection is created, the station selects the first and last hop satellites from the constellation [`StationUdlRouting`]. The station then performs call admission control [`StationQoSManager`] to determine whether there are enough resources [`StationResources`] to support the connection. Then the connection request is passed to the first satellite which computes the route [`IslRouting`] between the first and last satellites. If there is such a route, all satellites on the path perform call admission control [`SatelliteQoSManager`], [`SatelliteResources`]. The same procedure takes place in the destination station. If it turns out that the connection can be routed through that path, resources are actually allocated ([`StationQoSManager`], [`SatelliteQoSManager`]).

In order to simulate connections coming from a call concentrator (aggregated phone calls), the number of channels of the connection is not fixed after a connection has been set up, but can change during the lifetime of the connection. For example, a concentrator may set up a single connection for all the phone calls it handles, and may simulate both new phone calls and old closed phone calls by varying the number of channels used by the single connection as set up at start time. In other words, a number of *n* phone calls from station *i* to station *j* is simulated by the generation, in station *i*, of a unique connection that requests *n* channels. The modification of a connection requirements is performed in a similar way to the connection setup procedure.

A *handover* (or hand-off) occurs when either a UDL connecting a satellite to a ground station is cut off, or when a beam change occurs (inside the same UDL), or when an ISL is cut off. All connections passing through the affected link must be appropriately processed (rerouted or torn down) [`ConnectionChangeMonitor`].

A *connection drop* occurs when an existing connection is forcibly torn down. It may happen either when there is a handover and the connection cannot be rerouted, or when higher priority traffic preempts all the resources used by a connection. A *partial drop* may also occur, when part of the resources of the connection is taken back by the network. A *call block* occurs when a new connection cannot be established. It may happen when there are no resources available in the network in order to support the new connection. At the current time, the limitations of the call connections are: only point-to-point connections are considered; a connection cannot be split on more than one path (however, forward and return channels are not necessarily on the same path); no rerouting of connections happens as a consequence of growing or shrinking a connection (aggregate connections case); and no partial rerouting of connections is possible inside the constellation.

4.3 Routing

Routing policies are one of the main aspects that will be studied using Galileo. As mentioned in the last Subsection, routing is split into UDL routing and ISL routing. Up-Link (UL) routing is the process by which the source ground station selects the source satellite used to forward the packets of the connection, while Down-Link (DL) routing is the process by which the destination ground station selects the destination satellite from which the packets of the connection will arrive. Given a source satellite and a destination satellite, as provided by UDL routing, ISL routing computes the (or at least one) optimal path between these two satellites. ISL routing includes a signalling scheme [`SatelliteLinkStateManager`] to distribute and gather routing information [`RoutingInformation`] to/from the other satellites. End-to-end routing is therefore made up by UDL plus ISL routing.

4.4 Fault management

The general reliability of a satellite must cope with the reliability of each element as well as the relationships among different failures. Trying to compute the reliability function of a system is thus quite complex and many simulations have to be used in order to have an estimate of the reliability function. Galileo by itself, will not compute the reliability function since Consim is dedicated to this. Rather, the simulation engine of Galileo will be fed with events notifying failures. The nature and time distribution of these events is provided by Consim.

5 Some implementation aspects

This section will cover some implementation issues related to simulation performance. As it is often the case with broadband network simulations, the time needed to simulate a short period of time may be in the order of days; hence, the concern about performance enhancement. We will go briefly through considerations about simulating the network packet flows, distributing the simulation, programming optimisations and the selection of an appropriate development tool.

Simulating the actual packet flow in a network simulator provides valuable insight on the network behaviour. Without doing so, a satisfactory level of accuracy, especially when it comes to time relations of the various phenomena occurring in the network, cannot be achieved. Unfortunately, it also results in a heavy process (if not intractable) for a simulator of LEO constellations because of the potential huge number of traffic sources, and because of the bandwidth ranges involved (up to hundreds of Mbit/s). As a result, simulating each packet individually is often only a wishful thinking for realistic simulation scenarios. Two solutions are available to overcome this problem. The first solution consists in using mathematical tools (when it is possible) to model the average behaviour of the packets and deduce useful measures. For example, if the traffic is made of a number of constant bit rate sources, one can - given certain assumptions - model the cell arrival pattern in a switch using a $ND/D/1$ queue [**?**].

The second solution is to implement distributed or parallel simulation in order to multiplicate the available processing power [**?**]. Galileo plans to support both solutions. The simulation engine of Galileo was designed in order to ease the transition to a distributed paradigm without compromising the existing architecture.

Implementing distributed simulation raises two issues. The first one is how to partition the processing space into parallel processing entities. The second one is implementation related and concerns the communication means that are used among processing entities. As far as Galileo is concerned, one possible partition is to distribute evenly the satellites and stations on the pool of available computers. In order to choose a suitable partition, each possible solution must be evaluated taking into account the amount of data that has to be exchanged between the various distributed entities, the balance of the computation load on the different entities, the time dependencies between the entities and the available resources.

Once a distribution scheme has been established, the communication means must be chosen. Commonly such a mechanism provides *remote function call* like services. Java supports a distribution paradigm through remote method invocation. In a medium term range, the simulation engine of Galileo and LeoSim have been scheduled to make use of the RMI or other facilities (such as MPI) provided by Java. A survey of the different solutions available as well as of their performance has still to be performed. Currently, the simulation engine optionally supports parallel event processing on multiprocessor computers.

In a sequential or distributed simulation environment, performance improvements can be achieved at the implementation or system level. Enhancements are either related to the algorithms and data structures or to the development tools. All algorithms and data structures which are likely to be used often during the simulation must be carefully chosen. The agenda in the simulation engine is an example. Since thousands, if not millions, of events will be generated, queued and processed during a simulation run, these operations have to be efficient. As far as data structures are concerned, Java provides a library of core classes such as linked list, dictionaries or hash tables. One advantage of such libraries is that they are executed in native code (as opposed to byte-code). However, because these classes are designed to be as general as possible (regarding the type of objects they might store or whether the accesses might be concurrent), it results in performance impairements . When Java 1.1 was released, issues had been raised regarding excessive allocations, inefficient synchronisation or poor implementation in the core classes. Fortunately, these issues are addressed as time goes on.

Additional concerns are also raised by the nature of Java memory management which uses a garbage collector. Although garbage collection makes it convenient to write code less vulnerable to memory related bugs, this feature calls, during the implementation, for a careful attention of the object lifetime. Among other things, favouring object reuse is crucial in order to minimise the number of allocations as well as the number of objects eligible for garbage collection. This problem has surprising ramifications: past experience shown that LeoSim's execution speed has almost doubled by increasing the heap size, therefore reducing the number of times the garbage collector is invoked.

Java was initially a language for developing Internet applications and delivering them on different hardware architectures without recompilation. Compiling a Java program produces an intermediate language called byte-code. When the Java program is executed, the Java Virtual Machine (JVM) interprets the byte-code. The JVM takes care of the mapping between the byte-code and the native host architecture. Nevertheless, Java can also be used to develop applications that do not require seamless cross platform execution. The byte-code interpretation phase is a drawback from a performance standpoint. The first solution is to translate directly a Java source in native machine code. The GNU Java compiler (*gcj*) (still in development) provides such a facility. An intermediate solution is two use a JVM with a Just-in-Time (JIT) compiler that translates byte-code to native code upon class loading. Some measurements made with LeoSim showed that the increase in execution speed approaches 90%. These measurements were made using IBM's JDK under Linux. Other tests are carried out with Sun's HotSpot, and Symantec's JVM.

6 Galileo project management

Galileo is a medium-sized project with several remotely located teams participating. An effective mean to exchange information is mandatory. Furthermore, as for all developments, a structured approach is required. Galileo's project life cycle is following a spiral approach based on a core simulator incrementally enhanced. The analysis and design rely heavily on diagrams as a universal communication medium. The diagrams follow the UML standard and internal guidelines.

The first stage of the project consisted in writing in plain text what were the objectives of Galileo and ordering them by priority. Then the interactions between the user and Galileo were roughly described (using interaction diagrams from UML [**?**]). Using these diagrams as a starting point as well as our previous experiences, the system was described in terms of collaborating objects (i.e. objects exchanging messages). Then, these objects were grouped in classes. The class descriptions consisted in Java stubs documented using the *javadoc* utility from the JDK. At this point, Galileo was already a program compiling successfully, although without any processing done. This approach made it possible to gradually fill the gaps (i.e. replacing stubs with method bodies) while being able to test almost immediately the resulting code.

All deliverables are available in HTML from a Web server. Similarly, the source code is stored in a Web CVS repository. The CVS repository takes care of the versioning and is a useful tool to determine the changes made by different parties across successive versions. Currently, the primary development and analysis platform is Linux. All applications that were used during the design (*tgif*) and the development (*JDK*, *CVS*, *cvsweb*) are available free of charge.

7 Project status

Galileo was initiated in September 1998. Until June 2000, six Short Term Scientific Missions were organised and funded under the Cost253 action budget. Two additional missions were funded by the CNUCE-CNR. Galileo progresses mostly during these missions since the people involved (approximatively 2.5 persons from CNUCE-CNR (IT), ENST (FR) and Public University of Navarra (ES)) have their regular activities to carry on.

Currently, an initial version of Galileo is available with simple but operational components. Among them, a shortest path ISL routing algorithm, a resources management scheme using firm allocation, a call generator using Poisson arrivals and a handover resolution policy implementing complete rerouting. These components help to validate the Galileo architecture and, although they implement simple tasks, they provide some insight on the whole network behaviour. The effort is now put on providing Galileo with realistic components as well as setting up a testbed.

8 Conclusions

Considering the questions still open in the field of LEO constellations, there is an urgent need for a simulation tool that would provide means to study these questions. Galileo is meant to be this tool and will, as a first step, be aimed at the study of constellation access techniques, routing algorithms, and fault management. Galileo is an ambitious project with many challenges which will provide in the end a valuable tool for the organisations involved in LEO research.

References

[1] M. Annoni, S. Bizzarri, and F. Faggi. Performance evaluation of satellite constellations. the CONSIM(TM) simulator concept and architecture. In Springer-Verlag, editor, *Third European Workshop on mobile/personal Satcoms (EMPS'98)*, Venezia (IT), Sept. 1998.

[2] R. Brown. Calendar queues: A fast O(1) priority queue implementation for the simulation event set problem. *Communications ACM*, 31:1220–1227, 1988.

[3] N. Celandroni, E. Ferro, and F. Potortì. A simulation tool to validate and compare satellite TDMA access schemes. *Telecommunications Systems*, 12(1):21–37, 1999.

[4] K. B. Erickson, R. E. Ladner, and A. LaMarca. Optimizing static calendar queues. In *35th IEEE Annual Symposium on Foundations of Computer Science*, pages 732–743, Santa Fe (US-NM), Nov. 1994.

[5] E. Ferro. Proposal for a simulator architecture. Cost253 Temporary Document 10, CNUCE-CNR (IT), 1998.

[6] L. Franck. Leosim: a routing simulator for leos. Cost 253 Temporary Document 15, Brussels University (BE), 1998.

[7] R. M. Fujimoto. *Parallel and Distributed Simulation Systems*. Wiley, 2000.

[8] A. Muller. *Instant UML*. Wrox, 1997.

[9] J. Pitt and J. Schormans. *Introduction to ATM design and Performance*. Wiley, 1996.