GaliLEO

A SIMULATION TOOL FOR TRAFFIC ON LEO SATELLITE CONSTELLATIONS

PRESENTATION OF THE ARCHITECTURE

Nedo Celandroni [(*)], Erina Ferro[(*)], Francesco Potorti' [(*)], Laurent Franck[(+)]

[(*)]Affiliation: CNUCE/Institute on National Research Council (C.N.R.), Pisa, Italy
tel.: +39-050-593-207/-312/-203; fax.:+39-050-904052
E_mail: {n.celandroni, e.ferro, f.potorti}@cnuce.cnr.it

[(+)] Affiliation: ULB, Computer Science Department, Brussel, Belgium
tel.: +32-2-6505592; fax.:+32-2-6505609
E_mail: lfranck@ulb.ac.be

ABSTRACT

The aim of this work is to create a tool that can simulate the transmission of both connection-oriented and connection-less traffic over a constellation of LEO/MEO (Low Earth Orbit/ Medium Earth Orbit) satellites. We hope to provide a performance evaluation of various constellation access techniques and routing policies. The simulator, named GaliLEO, will be written in Java and will integrate results from another simulator (CONSIM from CSELT, I) to study the impact of faults in system performance.
GaliLEO should eventually become the simulation tool for the Cost253 action, thus providing a common tool for all studies taking place in the action. At the time of writing, considerable interest is being shown in GaliLEO.

INTRODUCTION

The initial design of the GaliLEO architecture was the outcome of a collaboration between two institutes where researchers had already had experiences in developing simulators. GaliLEO is the result of separate experiences carried out on two different simulators, LeoSim and SimToc. These two tools did not include all the issues we wanted to study. Moreover, other tools on the market only study some particular aspects of the transmissions on the LEO satellite constellations, or are simply too costly to maintain. Hence, the creation of GaliLEO which aims to develop a general purpose and customisable tool, freely available for the academic world.

LeoSim, one of GaliLEO's ancestors, is an event-driven, continuous time simulator written in Java and developed by ULB (B) in order to study a specific topic: link state routing algorithms in LEO satellite constellations. It relies on object-oriented techniques in order to be easily adapted to various routing algorithms. LeoSim provides statistics on the number of call requests, the call block probability, and the cost introduced by maintaining the link state database. LeoSim is still under development; however there is already a working version which supports static constellations and basic inter satellite link (ISL) routing algorithms. LeoSim also includes a user graphical interface. Dynamic constellations, handover management and elaborate end-to-end routing algorithms are to be implemented in the near future. Its design approach, as well as the simulation engine, has been transported into GaliLEO.
SimToc, the other ancestor, was more relevant to the ground station and to the up-down link (UDL) between the ground station and a satellite of the constellation; however its architecture is no longer being developed and all the efforts are now on GaliLEO.
Another simulator, CONSIM[(1)] developed at CSELT (I) for performance evaluations of satellite constellations affected by sudden partial failures, will be integrated "by results" into GaliLEO. Integration by results means that the two simulators are kept separate, and the results of one program are utilised by the other one. This is probably the best way to accommodate two simulators that were written separately, and to minimise 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 independent. In this case, CONSIM will be used in order to produce a list of faults which, according to the model, will occur during the constellation's lifetime. Each of these failure events specifies the type of failure as well as the time of occurrence. This list will then be used to feed GaliLEO's simulator engine in order to trigger the right fault managers at the appropriate time.

THE ARCHITECTURE

The main components of the GaliLEO architecture are shown in Fig. 1.

[(1)] CONstellation SIMulator

INPUT consists of the input files that contain the simulation run specifications (i.e., ground stations, traffic patterns, channel allocation policies, constellation characteristics, …etc). *VALIDATOR* performs the syntactic and semantic validation of the input data. *SCHEDULER* analyses the event queue and executes the actions relevant to the scheduled events.

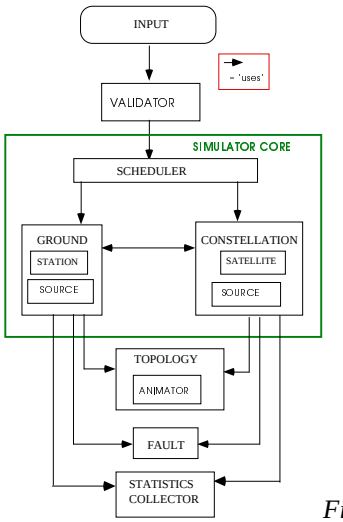

Fig. 1. Main components of GaliLEO architecture

GROUND handles the ground and the UDL aspects of the transmissions; it contains the *ground station* entity (Fig. 2).

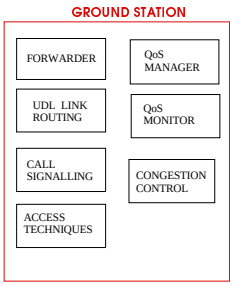

Fig. 2. The ground station entity

CONSTELLATION handles the ISL routing aspects of the transmissions; it contains the *satellite* entity (Fig. 3).

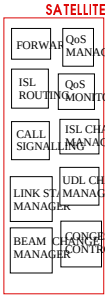

Fig. 3. The satellite entity

TOPOLOGY keeps tracks of the ground station and satellite movements; it contains the *animator* entity.

FAULT manages the failures which may happen to any of the elements of the simulated system. *STATISTICS COLLECTOR* collects, for each simulation run, the data necessary to compute off-line statistics required by the user.

Assumptions and definitions

GaliLEO is based on some assumptions and definitions, the most significant of which are reported here. A *cell* is an area of the earth illuminated by one satellite spot beam. A *footprint* is the whole coverage area of a 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 (i.e. a single subscriber or a concentrator) can receive a signal with an acceptable power level from more than one adjacent spot beam. A UDL is the aggregation of all spot beams pertaining to the same footprint; it has a fixed capacity, and is uni-directional. A *beam* is the communication medium between a satellite and a spot on the ground. A beam has a variable capacity which must not exceed the capacity of the UDL the beam belongs to. A *node* of the network is any station or any satellite. Satellites have multibeam antennas for up-link reception and down-link transmission, and are connected to neighbouring satellites by means of inter-satellite links (ISL) which are uni-directional. 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 that link must be re-routed. Connections are assumed to be full-duplex, with forward and return channels, where forward channels are intended to be from source to destination, and return channels from destination to source. A call connection drop occurs when an existing connection has to be dropped. It may happen either when there is a handover and the connection cannot be re-routed, or when high priority traffic preempts all the resources used by a connection. 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.

A satellite is associated with a *space position* which varies deterministically over time. Satellite movements are described through orbital mechanics; a ground station is associated with a *ground position* which may vary randomly over time if the station is mobile.

Traffic generators can generate both call connections and data. Data can be transmitted over connections (connection-oriented traffic) or in best-effort mode (connection-less traffic). In the following, when *data* is used, we refer to both connection-oriented and connection-less data, even though the simulation of connection-less data transmission over a LEO constellation will be not implemented during the first stage of this project.

Logical behaviour

Figure 4 shows the logical diagram of GaliLEO's behaviour.

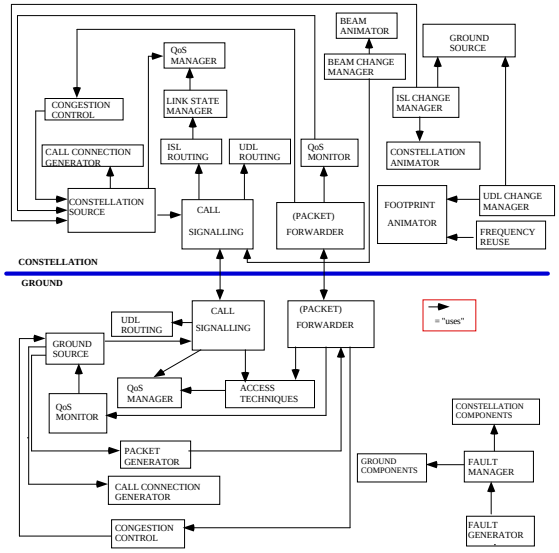

Fig. 4. Logical diagram of the architectural core of the simulator

The *ground source* module provides a model of the behaviour of ground traffic sources. It provides possible actions such as a request for a new connection, a request for releasing an existing connection, and a request to modify the requirements of an existing connection. *The packet generator* module implements the traffic models. The *UDL routing* module selects the first and the last satellites to enter and to exit the constellation, respectively. The entry satellite is chosen with respect to geometrical as well as to traffic load considerations, while the exit satellite is chosen with respect to geometrical considerations alone. The access techniques module handles the assignment of slots within a MAC frame. It is provided with information about the traffic (e.g. load and traffic type) and computes the band assignments accordingly.

The *footprint animator* module tracks the state of the UDLs between all satellites and all ground stations. In other words, the footprint animator is responsible for saying which satellite is able to communicate with which ground station. The *call signalling* module is responsible for establishing, releasing and modifying the connections. All these operations involve interactions with the equipment, or more precisely with the resource manager of the equipment. The *call generator* module provides a model for the dynamics of the connections. It gives information such as the connection requirements, the delay between two connections, the delay between two modifications of the connection requirements, the modified requirements, the source and destination of the connection, etc. The call modification facility is used to model the dynamic behaviour of a concentrator with a variable number of incoming connections multiplexed onto a single outgoing connection. In the simulation there are as many call generator types as types of users. The *forwarder* module provides services to switch the traffic packets. The switching is done either using a connection identifier (for connection-oriented communications) or using default routes (for connection-less communications).

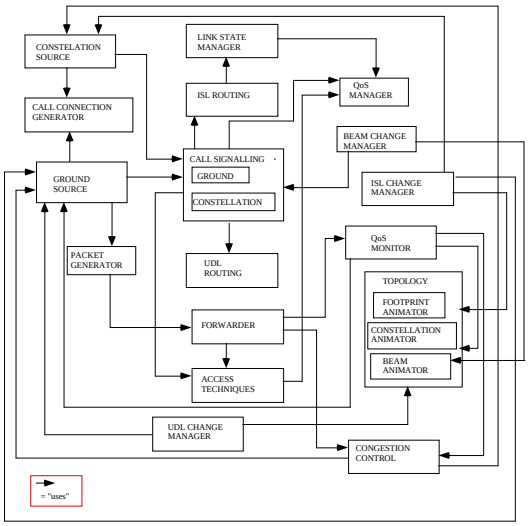

Fig. 5. The architectural core of the simulator from an implementation point of view

The *QoS (Quality of Service) manager* module is responsible for the management of the equipment resources as well as the translation from one requirement/resource description to another. The QoS manager also performs call admission control. There is one QoS Manager attached to each satellite and ground station. The *QoS monitor* module ensures that packets relevant to a given connection are compliant with the connection QoS contract. The *congestion control* module monitors the resource status in the station or in the satellite as a result of traffic. If congestion occurs, appropriate actions are taken such as throttling down the sources. The *frequency reuse manager* module takes care of frequency allocation for each spot beam of a satellite. The *ISL change manager* module handles all changes in ISL characteristics: this may happen in connection re-routing if, for example, an ISL is switched off. The *UDL change manager* module plays the same role as the ISL change manager for up and down-links. The *fault manager* module implements the fault reaction model. Notifications of faults are generated by the fault generator (this entity is an interface with results provided by CONSIM) and sent to the appropriate satellite components.

The *constellation source* module is similar to the ground source with more limited capabilities. The aim of the constellation source is to generate traffic confined within the constellation in order to provide an existing network load when ground sources are used. The *ISL routing module* computes the connection route within the constellation, between the entry point satellite and the exit point satellite as provided by the UDL routing module. Since the ISL routing algorithms used are link state, a link state map must be maintained in each node, which reflects the state of the whole network topology. In order to construct the link state map, each node broadcasts information about its status and gathers information broadcast by other nodes. It is the task of the *link state manager* module to handle the broadcasting and gathering of information and to maintain the link state map. The *ISL animator* module implements the orbital mechanics and therefore the satellite movements. The ISL properties are updated according to the satellite positions. The *beam animator* module tracks the state of a single beam within a given footprint. It detects the occurrence of changes in length occurring in the beams during satellite movement as well as the beam hopping from spot to spot. The *beam change manager* module reflects on the satellite the changes in beam configuration as detected by the beam animator. The beam change manager invokes, for example, rerouting if the result of the beam change is such that a connection can no longer be supported on that beam.

Figure 5 shows the above modules organised from an implementation point of view.

THE CONNECTION SET-UP

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 connection set-up model is synchronous, so that routines called in sequence allocate the resources (according to their availability) in a simple way. This makes it impossible to analyse the connection set-up time, as this model makes it instantaneous. The connection set-up procedure is triggered by a

generator, which creates a connection and passes it as an argument to the set-up routine of the source station. This routine returns a connection whose requirements are less than or equal to the one originally created by the generator. If no call block occurs, the returned connection is the same as the one passed as an argument to the set-up routine; otherwise it is reduced by an amount equal to the blocked channels. The connection may also get completely blocked due to a lack of resources. The set-up routine in the source station is the same as the set-up routine in each node, that is, in each station and each satellite. When a set-up routine in a given node is called with a connection as its argument, the node first checks for local availability of resources. If the connection can only be partially accommodated because of limited resources, the number of forward/return channels in the connection is reduced accordingly. The connection is then passed as an argument to the set-up routine of the next node towards the destination. When the set-up routine of the next node returns the possibly shrunk connection, local resources in the node are then allocated for the returned connection, i.e. the state of the node is updated, and the connection is returned to the caller.

The call connections are full-duplex, and variable in size. Connections may shrink either as a consequence of a handover, or because of the pre-emption of resources (for example, when high priority traffic has to be transmitted). At the current time, the limitations of the call connections are: the UDL routing is purely geometric and does not depend on congestion on the satellite constellation; end-to-end integrated routing is not supported by the design; 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 re-routing of connections happens as a consequence of growing or shrinking a connection (aggregate connections case); and no partial re-routing of connections is possible inside the constellation.

THE TOPOLOGY

One of the most delicate points in the GaliLEO architecture is the *topology* module. It includes the three sub-modules already mentioned: the *footprint animator*, the *constellation animator* and the *beam animator*. The functions of the topology module can be split into three domains: the *constellation topology*, defined in terms of satellites and ISLs, the *ground topology*, defined in terms of ground stations, and the *UDL topology* (Fig. 6).

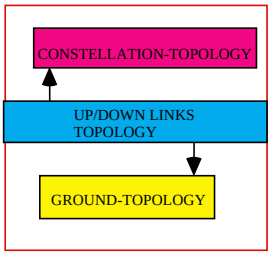

Fig. 6. The topology module

All these topologies are dynamic because satellites move with respect both to the ground stations and to each other; ISLs are switched on/off due to pointing/acquiring/tracking requirements; up- and down-links are also switched on and off; and ground stations may be mobile.

Each topology must have space and time references. The space reference(s) may be discretized. Although it is not of special use for the constellation, in some cases, as for example in Teledesic, the discretized space reference can be adopted for the ground topology because of the satellite spot-hopping capability. In this case, the discretization is performed by dividing the earth's surface into cells. In the UDL topology, since all references are already expressed in terms of ground stations and satellites, space discretization is not an issue. Although it is also possible to express the UDL topology co-ordinates in terms of ground co-ordinates and satellite co-ordinates (latitude & longitude positions), for the UDL topology we prefer a discrete space reference in ground station/satellite units. For the constellation topology, we decided a space reference for example in latitude/longitude units. Note that the time/space relationship involves both satellite movements and the earth's spin. Latitude/longitude co-ordinates are sufficient as long as satellites are considered to be at the same (or at least a close) altitude.

The constellation topology

This module generates events for all interesting state changes in the ISLs. In particular, an event is generated when a link goes on or off, and an event is generated when a link's length changes by more than one user-specified threshold. The event trigger handling of the changes is managed in the ISL Change Manager. It is responsible for checking that those connections which are using the modified ISL are still valid. If not, connection re-routing is performed.

In order to work properly, the constellation topology needs to know the trajectory of each satellite (usually done by describing the parameters of each orbit), and the initial position of each satellite.

The ground topology

The ground topology returns the time-dependent position of a ground station (the co-ordinates

depend on the space reference chosen), and the position of a ground station with respect to another ground station (information used for frequency reuse computation). It needs to know the initial position of each ground station, and the trajectory of each ground station which is assumed to be mobile.

The UDL topology

The UDL topology covers both the dynamics of the footprints (footprint animator) and the dynamics of the beams within a given footprint (beam animator). The Footprint Animator generates events to the satellites and the stations when the situation of a station changes with respect to a given footprint (for example, if a station leaves the footprint of a given satellite or if a station enters the footprint of another satellite). Using this method avoids having to poll the footprint status. The Beam Animator generates events which have to be placed in the context of a given footprint. It notifies the satellite and the station that some beam characteristics have changed (this case is likely to happen if steerable antennas are used) or that a station is changing beam (in the case of fixed antennas).

ROUTING

Routing policies are one of the main aspects that will be studied using GaliLEO. In the LEO constellation context, routing is usually 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 routing is the process by which the destination ground station selects the destination satellite from which the packets of the connection will arrive.

The criteria used for UDL routing is the availability of resources in the satellite and in the ground station, the minimisation of the handover rate on the UDL, and the quality of the communication between the ground station and the satellite.

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. The criteria used are: resource availability in the satellites and ISLs, minimisation of the handover rate, quality of the communication among satellites, and length of the path.

Once UDL and ISL routing have been defined, we can define the end-to-end routing as made up by UDL plus ISL routings. Whether end-to-end routing should be considered as three separate processes or one whole process (referred to as *integrated routing*), is an issue which will be studied in GaliLEO. Non integrated end-to-end routing results in possibly non optimal routes, while integrated routing leads to scalability issues.

Let us consider the configuration depicted in Fig. 7.

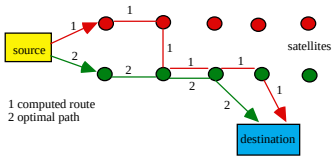

Fig. 7. An example of a configuration

Considering UL/DL routing as separate processes, let us suppose that the up-link selected is the one denoted by ‘1’ between the source and the constellation. The ISL routing must then compute an optimal path between the source and destination satellites, as chosen by the UL and DL routing. However, if we consider end-to-end routing as a single process, the up- and down-links selected might not be the same (as denoted by the ‘2’). Although they are not optimal in the context of U/D link routing alone, they belong to the optimal path in the context of an integrated end-to-end routing.

Integrated end-to-end routing can be implemented in the ground stations or in the satellites. In the first case, the ground stations must have information concerning the whole topology, i.e. the link state maps, which contain information about the availability of resources in each satellite and in each ISL, in each ground station and in each up/down-link. Since there are more ground stations than satellites, keeping this information up to date is a heavy task; moreover the information must be conveyed through the constellation and distributed to each ground station. As a result, performing routing in the ground station does not completely remove the routing functionality from the satellites. The broadcasting and processing of link state information must still be performed in each satellite. On the other hand, the route computation is only present in each ground station. Apart from this aspect, the routing is a straightforward process with minimal interaction between ground stations and satellites.

If implemented in the satellites, there is a price to pay in an additional complexity in the routing process, but the information needed does not have to be sent to each ground station. As one of the goals of routing is also to determine which up-link to use, when a ground station is asking for a route to a satellite, it is not able to determine which satellite the request must be forwarded to. One solution is to send the route request to all the satellites currently in view. These satellites then have to compute the optimal route and send the responses to the ground station. The station then selects the actual optimal route from the proposed set of routes. This is quite heavy but it minimises the information that has to be stored in the ground station.

Figure 8 represents the interaction between routing and other components of GaliLEO inside the ground station.. It is assumed that integrated end-to-end

routing is used, and that routing takes place in the ground station.

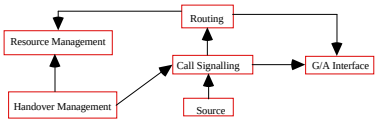

Fig. 8. The situation in the ground station

If we consider a reference model as introduced by an OSI stack, we can draw an equivalent 'layered' view as represented in Fig. 9. The higher a box is in the picture, the closer it is to the application layer. The lower a box is, the closer it is to the physical media.

Figure 10 shows the equivalent layered view for the satellite (the non layered view can easily be derived from).

The service provided by the routing component is to compute a route which supports a connection characterised by a source, destination and user requirements. Furthermore, routing needs a resource management component that provides information about resource availability, and an Air/Ground and Air/Air interfaces for the routing algorithm to exchange information with other routing algorithms in the constellation.

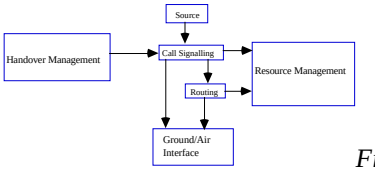

Fig. 9. Layered view

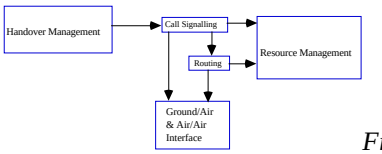

Fig. 10. Satellite layered view

FAULT MANAGEMENT

Today's satellites are getting more and more complex in order to support the current broadband services. Increasing the payload complexity leads to an increase in the number of possible fault sources and also a financial impact. Consequently it is crucial to accurately model the reliability of a satellite constellation. By *reliability* we mean the probability that a given failure does not occur in a certain time frame. Reliability can thus be completely described with a density function as defined in the probability theory. The goal of reliability modelling is therefore to select the proper density function for each element to be included in a model and to calculate the resulting reliability accordingly. Before trying to compute the density function, it is mandatory to give a precise description of the system (i.e. its architecture) for which we are trying to define a reliability model. Some assumptions have to be made. An obvious example is the use of ISL. It is relevant to the present analysis, since ISLs need dedicated hardware which has to be taken into account in the reliability model. The various faults that might occur can be roughly categorised into *partial* and *catastrophic* failures. While the former still allow the satellite to work (even in a degraded mode), the later are fatal. Each of these failures then has to be weighted over time using a probability density. The determination of these density functions can be complicated and necessitates an extensive understanding of how a satellite is made up. Finally, all these density functions (or their respective reliability functions) must be merged in order to compute the general reliability function of the satellite. Note that even if two failures are only partial, this can still be fatal to the satellite. Therefore the general reliability must cope with the reliability of each element as well as the relationships among different failures. Trying to compute this reliability function 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.

SOME IMPLEMENTATION ASPECTS

This Section will cover some implementation issues related to simulation performance. As 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 distributing the simulation, choosing carefully the right simulation tool, and simulating the network packet flows.

One of the performance issues of the GaliLEO design is the distribution of the simulation components among different computers. One possible partition is to run all the ground stations on one computer and the constellation on another. 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 resources available. Once a distributed framework has been established, the mechanism supporting the distributed paradigm must be chosen. Commonly such a mechanism provides "remote procedure call" like services such as those with Sun's RPC or Corba architecture. It is also crucial that the transition from

local calls to distributed calls does not entail the software being completely rewritten. Both GaliLEO and LeoSim are written in Java and use the same simulation engine. Java supports a distribution paradigm through the definition of serializable classes and through remote method invocation. These two mechanisms are often referred to as RMI (Remote Method Invocation) facilities. Remote method invocation occurs when a method is invoked on an object which is not stored locally but which is accessible through a network. The method call is then transferred to the target machine, is executed there, and the results are transferred back to the computer which called the method. In order to be invoked remotely, the class including these methods must by some means register the methods. The concept behind serializable classes is strictly connected to remote method invocation. Considering a remote method which accepts a complex variable as its parameter (i.e. an object, not a basic type), the parameter value must be transferred along with the method invocation. Furthermore, the target computer (which may be using a different hardware) must be able to understand the parameter value. A class that can be transferred in such a way is called a *serializable class*. Basically, the class must provide a set of rules defining the coding and decoding of the instances of this class, and this results in an intermediate representation independent of the network architecture, as well as the source and target computers.

In a medium term range, the simulation engine of GaliLEO and LeoSim have been scheduled to make use of the RMI facilities provided by Java in order to distribute the simulation execution. Currently, the simulation engine optionally supports distributed event processing on SMP (Symmetrical Multiple Processing) computers.

Enhancements in system performance at an implementation level are either related to the algorithms and data structures or to the development tools. All algorithms and data structures which are likely to be called often during the simulation must be carefully chosen. This is the case for the handling of the event list. 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 the development tools are concerned, care must be taken about the execution of Java programs. Java was originally intended to be the language for developing Internet applications and delivering them on different hardware architectures without re-compilation. In that context, Java is an interpreted language making use of a Java virtual machine to execute byte-code resulting from the compilation. The Java virtual machine takes care of the mapping between the byte-code and the native host architecture. However, Java is also a classical programming language used to develop applications that do not need seamless cross platform support. This is even more true since the interpretation of byte-code results in severe performance degradation. In this case, care must be taken in using a Java virtual machine including a JIT compiler, which translates on-the-fly byte-code into native code. Preliminary measurements with LeoSim showed that the increase in execution speed approaches 90%. These measurements were made using a JIT compiler named TYA under Linux with JDK 1.1.6. Other tests must be carried out with Kaffe and Guavac (other JIT compilers under Linux) and with Symantec's JIT compiler bundled with Borland's JBuilder under Windows 95. There would also appear to be work in progress to develop Java GCC front end (Cygnus' Sourceware Project for Java) and also a Java to C translator (named Toba). Simulating the packet flows in a network simulator is useful because it is the only way to have a precise model of network behaviour. Unfortunately, this is also rather a heavy process, and even heavier for a simulator of LEO constellations because there is potentially a huge number of traffic sources (since the constellation covers the whole earth), and there is an enormous number of packets crossing the network per unit of time (since the bandwidth offered by the equipment is in the order of hundreds of Megabit/s). As a result, simulating each packet individually is just wishful thinking, so a way has to be found to provide a satisfying description of the packet flow behaviours without having to simulate them individually. Two possible solutions are: i) simulating packet batches; ii) modelling the effects of the packet flows and using this model in the simulation.

The first solution is equivalent to using a time scale coarser than when simulating packets individually. This approach is valid as long as the use of a less precise time scale does not lead to the loss of interesting events that have to be tracked individually. By using this technique, it is possible to decrease the number of events related to the packets by a ratio which is equal to the average size of a packet batch. This approach has the advantage of being widely applicable. The second solution is more easily explained with an example. Currently, in LeoSim, simulation takes place at a call level only. Although this approach is efficient, it does not simulate buffer overflows which might in turn result in connection drops. A rather optimistic approach is used, where the call admission control is expected to accept a connection only if no overflow due to that connection can ever happen. Another approach is to construct, in a pre-study, a model of the buffer's behaviour, by knowing the call admission control algorithm used, the state of the buffer, and the load offered to the buffer. The result of this pre-study is, for example, a two dimensional matrix providing a probability distribution for each of the

pairs (buffer state, offered load) for a given CAC algorithm. This matrix is then used by LeoSim in order to generate random buffer overflows. As the state of the buffer and the offered load change, a different cell in the matrix is used. If it turns out that this approach works, the performance increase would be even more interesting than the previous solution using packet batches. Unfortunately, the example presented here is a straightforward one since the model established only has to provide information about buffer overflows. It is likely that an elaborate traffic behaviour model taking into account all interactions occurring in the system will be difficult to obtain.

GaliLEO PROJECT MANAGEMENT

GaliLEO is a large project. Moreover, it is being developed by several teams working remotely. As a result, the project management of GaliLEO is crucial. GaliLEO's project life cycle is following a spiral approach based on a core simulator which is gradually being enhanced.

The analysis and design are object oriented since the usage of object orientation is one of the corner stones of GaliLEO's general nature. The methodology relies heavily on diagrams since the communication within the project team is one key point. Diagrams are following the UML standard as well some internal guidelines. All deliverables are accessible, in HTML, from a Web server. The same applies for the source code which is accessible from a Web CVS repository. Currently, the primary development and analysis platform is Linux and no commercial software is being used. We hope to be able to carry on like this.

CONCLUSIONS

Considering the questions still open in the field of LEO constellations, there is an urgent need for a simulation tool that would provide a 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. The development team of GaliLEO is located over many different places, and this implies the need for well defined analysis and design phases as well as a tightly managed development.

GaliLEO is an ambitious project with many challenges which will provide in the end a valuable tool for Cost253 activities as well as for the organisations involved in LEO research.

REFERENCES

[1] Simulation of a Routing Algorithm using Distributed Simulation Techniques; C.D. Pham, J. Essmeyer, S. Fdida.

[2] Reliability modelling for constellation systems; M. Annoni – CSELT; Cost 253 Working Document TD(98) XX

[3] Instant UML; P.A. Muller - Wrox Press.

[4] The Java Tutorial (2nd edition); M. Campione, K. Walrath, A. Huml; Addison Wesley.

[5] The Java Tutorial *continued*; M. Campione, K. Walrath, A. Huml; Addison Wesley.

[6] Concurrent Programming in Java ; D. Lea ; Addison Wesley.

[7] UML and C++; R.C. Lee and W.M. Tepfenhart; Prentice Hall.