Object-Oriented Design of Real-Time Telecom Systems

Jean-Marc Jézéquel

Object-Oriented Design of Real-Time Telecom Systems

Jean-Marc Jézéquel
Irisa/CNRS
Campus de Beaulieu
F-35042 RENNES CEDEX, FRANCE
E-mail: jezequel@irisa.fr
http://www.irisa.fr/pampa/PROF/jmj.html
Tel: +33 299847192 — Fax: +33 299847171

Abstract

Many engineers are still reluctant to adopt advanced object-oriented technologies (such as high modularity, dynamic binding, automatic garbage collection, etc.) for embedded systems with real-time constraints, because of their supposed inefficiency. We set ourselves into the context of building telecommunication systems with a standard object-oriented analysis and design approach. We describe how we use relevant design patterns, followed with an implementation in a pure object-oriented language (Eiffel) to conciliate the needed efficiency with the benefits of the object-oriented approach —flexibility, dynamic configurability, maintenability, portability, etc. We discuss a case study based on the implementation of SMDS (Switched Multi-megabits Data Service) servers featuring high-throughput and low-delay transmissions and respecting the real-time constraints of SMDS.

Keywords Real-time, Telecommunications, SMDS, OO Analysis and Design, Design Patterns, Eiffel

1 Introduction

Object-oriented technologies have made their way into numerous application domains ranging from the MIS world [21] to high performance numerical computing [6]. However there are still very few publications exploring the use of advanced object-oriented technologies for building embedded systems, and most particularly embedded telecommunication systems. Thus telecommunication engineers are still reluctant to go beyond the use of very basic data abstraction mechanisms (e.g., C++ as a better C), because more advanced concepts (such as high modularity, design by contract, heavy use of multiple inheritance and dynamic binding, exception handling, automatic garbage collection, etc.) are widely believed to be not efficient enough to match the real-time constraints that usually go with embedded telecommunication systems. However there is a growing interest in ripping more benefits out of these advanced object-oriented technologies because the versatility of the new value added telecommunication services induces huge software development costs that need to be paid off on a longer term (e.g., more than one hardware generation). Telecommunication engineers find themselves paying more and more attention to software engineering issues such as flexibility, dynamic configurability, maintenability, portability, etc., which often conflict with the traditional way of finely tuning the software to get the best performances out of a given architecture.

The aim of this paper is to show how an object-oriented analysis and design (OOAD) approach, augmented with a set of relevant design patterns and followed with an implementation in Eiffel[1] may be used to build a highly evolutive and portable Switched Multi-megabits Data Server (SMDS) server, still respecting the (soft) real-time constraints of SMDS and featuring high-throughput and low-delay transmissions.

2 Building a Distributed SMDS Server

2.1 The Switched Multi-megabits Data Service

The Switched Multi-megabits Data Service (SMDS) [2] is a connectionless, packet-switched data transport service running on top of connected networks such as the Broad-

[1] Eiffel [17] is a pure object-oriented language featuring multiple inheritance, polymorphism, static typing and dynamic binding, genericity, garbage collection, a disciplined exception mechanism, and systematic use of assertions to improve software correctness in the context of *programming by contract*.

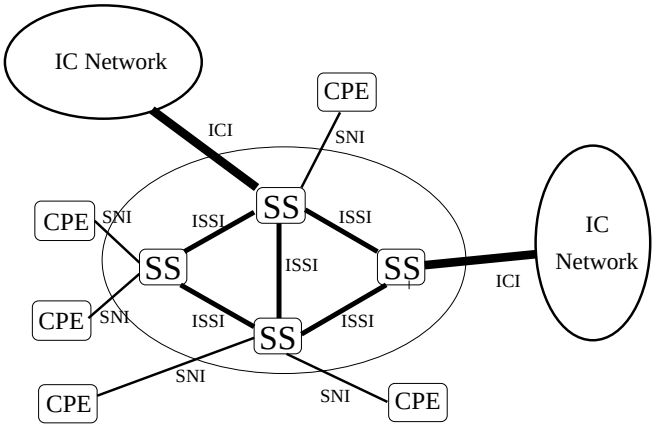

Figure 1. Architecture of an SMDS network

band Integrated Service Digital Network (B-ISDN), which is based on the asynchronous transfer mode (ATM).

SMDS was designed to provide high throughput and low-delay transmissions, and to be able to maintain them over a large geographical area. As a result, it can be used to interconnect multiple-node local area network (LANs) and wide area networks (WANs), and provide them with "any-to-any" service (a capability sometimes referred to as the *dial-tone for data*)[2] SMDS relies on an overlay network with non-ATM switching to transfer connectionless messages [13]. This network consists of a set of interconnected *connectionless servers*. Clients who are willing to do connectionless traffic have to access the nearest connectionless server using any available protocol, such as an ATM connection.

2.2 System Requirements

An SMDS network is based on a three-tiered architecture: a switching infrastructure made of SMDS Servers (SS), a delivery system made of SNIs (Subscriber Network Interface), and an external network access system, ICI (Independent Carrier Interface). So each SMDS server has to switch packets coming from SNIs, ICIs, and ISSI (Inter Switching-System Interface) links (see figure 1).

SMDS has been designed to be supported by various lower-level layers, e.g. ATM (AAL 3/4 or even 5) or DQDB (see figure 2).

[2] Now that it is possible to do IP over ATM in a standard way, the SMDS has lost a lot of its commercial interest. Still it makes an interesting case study from the technical point of view.

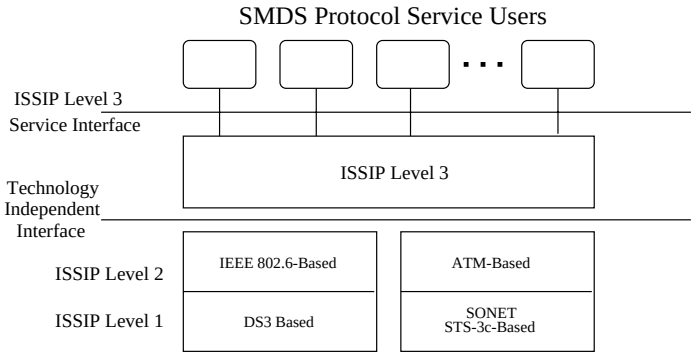

Figure 2. Architecture of an SMDS server

2.3 Real-time and Performance Constraints

2.3.1 Real-Time Constraints

Most telecommunication systems apparently have many real-time constraints. Among them, we can distinguish between latency/delay requirements and total throughput requirements. In the former case, we can find a requirement that an SMDS server must say "*hello*" to each of its neighbors at least every 10 ms, or else it can be considered as "*dead*" (the communication link has then to be reset). But this requirement, which is by the way the tightest real-time requirement in SMDS, is not really as "hard" real-time as it seems. Indeed failing it once in a while is not as big a catastrophe, as this could be the case in e.g., an aerospace project: here a re-synchronization protocol is just restarted where it would not have been necessary.

The later case, total throughput requirements, boils down to performance constraints. There are a number of studies in the literature identifying bottlenecks in high-performance communication systems, concerning most notably header processing speed, data movement inside the system, execution environment, and interface between the processing system and the physical layer.

2.3.2 Header Processing Speed

The packet header processing speed determines an absolute limit on the global performances of the communication system: the system may not have a throughput per I/O board larger than the maximum packet size divided by the header processing speed. This header processing speed has then to be optimized as much as possible. The introduction of parallelism has been considered at this level, but it does generally not pay off [22] because of the limited intrinsic provision for concurrency in such kind of processing. This tendency is even enforced with recent "light-weight" protocol (XTP, SMDS, etc.) featuring simplified header processing, actually leaving nearly no room for parallelization.

In our SMDS server, the significant figures are the speed of header processing in different contexts: transmission (packet received from an SNI and then injected in the SMDS network), reception (packet coming from an ISSI link and delivered to a SNI), and switching of traffic (from an ISSI link to another one). To reach Gigabit flow rates, all of these various processings must be handled within less than $0.5\mu s$.

2.3.3 Execution Environment

These processing times are directly proportional to the processing power of the processor: the execution environment has thus a major influence on performances. The raw power of each processor actually determines the throughput of the system, and the bandwidth of the channels connecting the different nodes limits the global data flow. If the software is portable, it can directly benefit from the very fast speed improvement of hardware components.

2.3.4 Data Movement inside the System

Another important source of performance loss is linked to data movement inside the system [25]. To be efficient, an implementation should avoid to copy packets from memory to memory because of its time cost (DRAM bandwidth does not follow the exponential increasing of processing power). Also, for parallel implementations, data transfers between two separate nodes should be minimized because excessive internal communications can lead to link saturations and even to a global system slowdown. A classical solution [4] is the use of a shared memory where all the packets are stored. The processing nodes would only access the headers and trailers of these packets and would never deal with the data they include.

2.3.5 Interface between the Processing System and the Physical Layer

The last bottleneck may be the interface between the processing system and the physical layer [4]: its throughput should be sufficient not to limit the system. If the number of access points to the physical links is insufficient, these access points are considered as shared resources. They can then be saturated if too many processing units try to use them at once.

The logical solution to this problem consists in handling various network access points in parallel. For that, we distribute the ISSI and SNI connections among the various nodes of a distributed computer (see figure 3), each one having its own OS and interface(s) with the SMDS network, and collaborating with other nodes to provide the SMDS service.

3 OO Analysis and Design

3.1 An Object-Oriented Approach

The implementation of a distributed SMDS server was a telecom, research-oriented project carried out in our lab [10]. We also used this case study in [8] to present some software engineering issues related the full usage of an object-oriented approach, from analysis and design down to an implementation in Eiffel. In this Section, we just outline the various steps of the approach, while concentrating on design issues related to the distributed real-time aspects of our SMDS server.

Most object-oriented methods now make it possible to tackle with the incremental, iterative, and evolutionary nature of software development. The various phases of analysis, design, and implementation use the same conceptual framework (based on objects) and have no rigid frontiers between them, so the object-oriented software engineering process can be called *seamless*. Seamlessness is so important for modern software systems because the main effort in software development (perhaps 80% or more) is spent not on new development but on maintenance of existing software. Therefore, the very role of analysis and design in software engineering is changing. Rather than addressing only the earliest stages of the software lifecycle, it is increasingly being viewed as the intellectual support needed across the entire software construction process.

3.2 Analysis through OO Modeling

The first step towards an object-oriented analysis is concerned with devising a precise, relevant, concise, understandable, and correct model of the real world. The purpose of object-oriented analysis is to model the problem domain so that it can be understood, and serve as a stable basis preparing the design step. Note that the requirements for many systems are rarely as formal as protocol specifications, often they evolve together with the design and implementation of the system. An important point of OO development is to make building such systems possible. It is often said that we don't need to understand the entire problem before starting to code —as long as we develop classes for the parts that we do understand.

The Object Modeling Technique (OMT) [18] is one of the most popular OO Analysis and Design method. The OMT analysis model extends itself in three dimensions:

- the object model, showing the static structure of the real world system through abstract or physical classes and their relationships.
- the dynamic model, showing the temporal behavior of the objects in the system. When the problem domain deals with telecommunication protocols, most

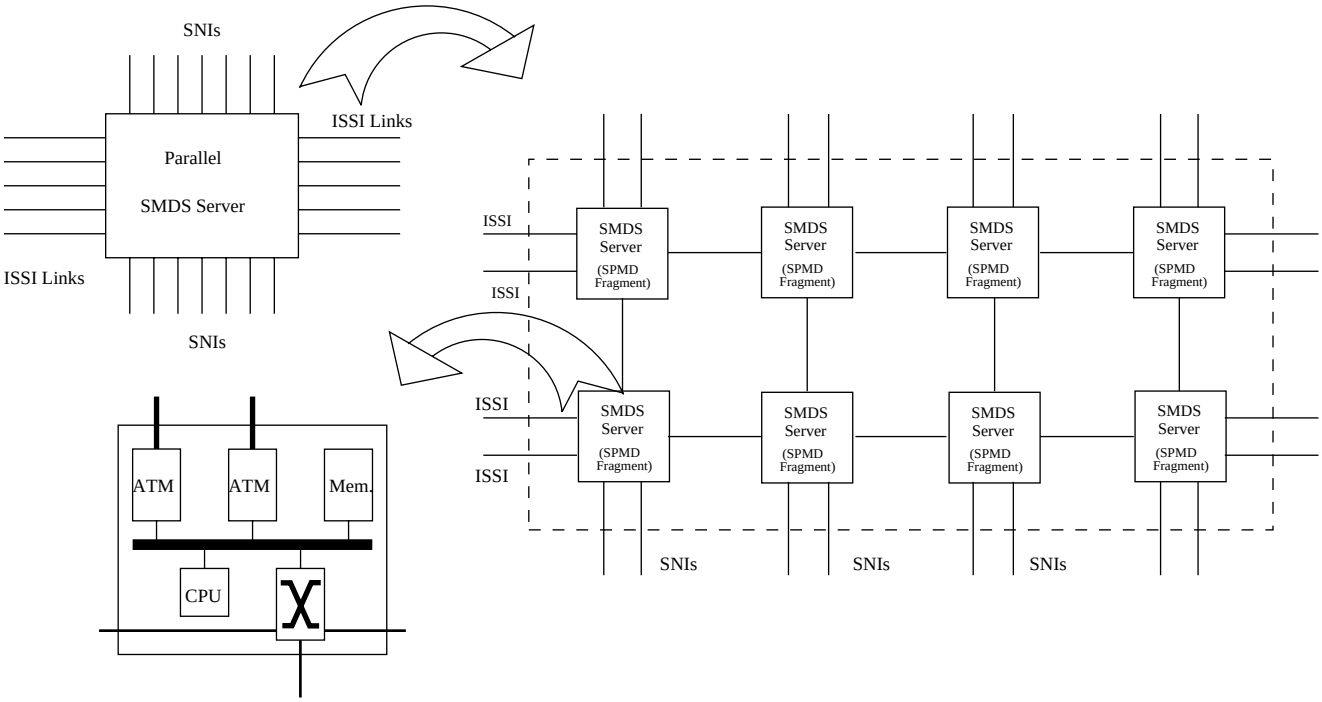

Figure 3. Distributing the network connections to get a Parallel SMDS server

of the analysis work regarding the dynamic model is already done, because it is explicitly included in the requirements as the definition of the protocol. This definition is usually made of chunks of communicating finite state automata, extended with some private variables and watchdog clauses.

- the functional model, showing the constraints between the objects in the system (and notably between inputs and outputs).

3.3 Design Patterns for High Performances

3.3.1 System Design

The *design* phase starts with the output of the analysis phase and gradually shifts its emphasis from application domain to computation domain: the implementation strategy is defined, and trade offs are made according to the priorities defined in the previous section.

For most real-world applications, the first step in system design is to divide the system into a number of components. These components should be rather autonomous and loosely coupled subsystems, featuring well-defined interfaces. The main subsystems of our SMDS server are the three levels of the ISSI, the ICI, the SNI, and the network management interface (NMI) (see Figure 2). Due to performance constraints, the ISSI levels 1 and 2 are implemented with a set of dedicated boards, responsible for handling the underlying protocol (ATM, DQDB, etc.). Whatever their type, these boards present the same interface to the ISSI level 3: this interface can be modeled with an abstract class, which is given a specific concrete implementation for each kind of board that can be plugged into the server (plus one for simulation purposes on the development station). In the following, we concentrate on the core of the SS, the part dealing with the communications between SMDS servers (that is the ISSI).

3.3.2 The Active Object Approach

In our SMDS server analysis model, as in most telecom systems, many classes have associated finite state machines (FSM) describing their dynamic behaviors: their instances are often called *active objects*. This notion of active objects was introduced in concurrent OO languages such as POOL-T [1] or ABCL/1 [23] to merge both notions of objects and processes. Active objects can be seen like concurrent activities communicating by sending messages. A classical approach is then to use a so called “real-time operating system” and allocate a real process to each active object. These processes must then be properly scheduled to meet real-time constraints, which can be helped by many formalisms and tools [3].

While conceptually simple and general purpose, this approach suffers from a number of drawbacks:

- inheritance anomaly: This simple parallelism model does not melt well with inheritance, because synchronization constraints are hard to inherit in a context in which subtype substitutability is to be preserved. This problem is known as the *inheritance anomaly* [14] and is not easy to circumvent [15].
- context switching overhead: when we have many processes, the context switching time can become significant. Light Weight Processes (e.g., *threads*) can help here, but it is still very difficult to handle hundreds or thousands of active objects with their own thread of execution, if only because many operating systems limit the number of threads per process.
- wasted system ressources: at a given point in time, most processes are simply waiting for something to happen. While they do nothing, they needlessly waste system ressources (stack space, etc.).
- unnecessary message buffering: handling a typical message (e.g., from ISSI to SNI) involves several processes that must exchange information. In most operating systems, it means that the information is queued, even if the target process is resident in the same processor.

3.3.3 The Reactive Approach

On the other hand, if the smallest granularity of time is around a few milliseconds, as it is the case in our SMDS server, many events can be fully handled at once, that is in an atomic way. For example, an incoming message goes up various levels of the protocol stack (implemented as objects) following a string of up-calls. At the relevant level, it is processed (e.g., the header is modified for the next hop into the network), and then sent down the stack still using method calls. For the sake of efficiency, a clever compiler may statically bind most method calls, and even inline them in the context of the caller: we keep the modularity of the protocol stack architecture at the design and implementation level, but the compiler is able to make cross-module optimizations[3].

Avoiding preemptive scheduling would allow the system to finish current work before beginning new work and thus remove many resource management and concurrency problems. The core of the SMDS server may then be an event dispatcher. It monitors all possible sources of events, serializes them, and dispatches them to the relevant handler objects for an atomic processing. The event dispatcher is implemented by *polling* the various event sources:

1. Commands coming from the "system manager" or from the operator.
2. *Timers* firing.
3. Remote method invocation coming from other processors in the same switch.
4. Message reception at one of the local network interfaces (SNI, ICI, ISSI).

This architecture is very close to the *Reactor* pattern that has been documented in [19], and whose intent is:

> *Support the demultiplexing and dispatching of multiple event handlers, which are triggered concurrently by multiple events. The Reactor pattern simplifies event-driven applications by integrating the demultiplexing of events and the dispatching of the corresponding event handlers.*

3.3.4 General Behavior

The general behavior of an SMDS server is illustrated by the Eiffel code snippet provided in Example 3.1. The initialization of our SS consists in creating the main entities and provides them with the references they need to establish the cooperations identified in the object model. We use an *Abstract Factory*[4] to be able to dynamically configure the variant parts of the SMDS server, e.g., how many ATM and or DQDB interfaces does it have (See [9] for more details on our approach at configuring object-oriented software). Once the initialization is completed, the SS enters the polling loop, whose termination may be required by the operator. Its effect is to finish the polling loop, clean up everything, and terminate the "main" program. Failures in the SS should be reported by means of exceptions; dealing with fatal failures consists of handling otherwise unexpected exceptions (non-fatal errors are reported as warnings to the Operator). The main procedure is provided with a *rescue* clause to handle these exceptions by cleaning things up and restart the server if the crash rate stays below a configurable threshold (the crash rate is simply a variable that is periodically decremented, *a la* leaky bucket). Unless the crash rate

[3] For example, GNU SmallEiffel is an Eiffel compiler which uses a fast simple type inference mechanism to remove most late binding calls, replacing them by static bindings [24]. Since the whole system is analyzed at compile time, multiple inheritance has no overhead at all. Efficient dead code removal makes small executable files and increases performance of the compiler itself. SmallEiffel features an original coding scheme to eliminate the need for virtual function tables, thus using modern hardware more effectively. This new implementation of dynamic dispatch does not break control flow. Furthermore, this allows the in-lining of more calls even when dynamic dispatch is required. The advantage of this approach is that it greatly speeds up execution time and considerable decreased the amount of generated code.

[4] Abstract Factory is one of the Creational Design Patterns documented in [5]. Its intent is to provide an interface for creating families of related or dependent objects without specifying their concrete classes.

is too large, and the server failure cannot be considered transient (in which case it is simply stopped), the *retry* instruction works like a kind of warm boot, because everything restarts as if it were initialization time. The garbage collector then will recycle the resources associated with the overridden entities. Like many telecom protocols, the SMDS has been designed to deal with server crashes (fail stop) and later recovery, so this simple mechanism also provides a limited amount of both hardware and software fault tolerance. It also may be disabled during the testing phases to get a standard exception history dumps.

Example 3.1

```
main is
        -- initialize and run this SMDS server
    do
        from initialize
        until is_shutdown_required
        loop
            process_next_event -- by polling event sources
        end -- loop
        clean_up
    rescue
        clean_up
        crash_rate := crash_rate + 1
        if crash_rate <= crash_rate_threshold then
            retry
        end -- if
    end -- main
```

Because the system spends much less time in managing the processes and their conflicts, the reactive approach is also much more efficient that the one based on the use of active objects. However, if we adopt this approach, we have still to deal with three kinds of problems: (1) remaining asynchronous interrupts, (2) the case where the handling of an event takes too much time to be processed atomically and, because we used an advanced OO language, (3) garbage collection.

3.3.5 Asynchronous Interrupts

Asynchronous interrupts still arise due to e.g., timer ringing. Indeed, at any given point in time, many watchdogs are active in most telecom systems. Because of their number, we implemented them in software, using a class TIMER featuring an expiration date, and the object to ring when the date is reached. The timers are stored in a priority queue (by increasing date of awakening). On loading (or disabling), a timer inserts (or removes) itself in (or from) the queue. Polling this queue at each iteration of the main loop revealed itself too costly, so we used a hardware timer set to fire when the software timer at the head of the queue is about to reach its expiration date. This hardware timer uses an asynchronous interrupt to notify the server, but we do not want to disrupt the current processing of an event in the server. So we set up a simple interrupt handler to only set a "timer is ringing" flag. This flag is checked in the main polling loop, and thus the timer ringing event is processed in the next iteration.

3.3.6 Events that Require a long Processing

Once in a while, the processing of an event can take longer than the maximum time allowed for an atomic processing (around 5 ms in our SMDS server). In this case, we require that its handler periodically call the *process_event* scheduling primitive to allow a handler with a finer granularity to be run. Because 5 ms is a large amount of time for a multi-MHz processor, this is usually a simple matter of placing the scheduler call at the end of each relevant loop in the handler code. Each event handler method has thus an associated postcondition stating that less than 5 ms must be spent on its execution. If this postcondition is violated, it denotes a bug in the handler code.

3.3.7 Subduing the Garbage Collector

With many languages, programmers must explicitly reclaim heap memory at some point in the program, by using a *free* or a *dispose* statement. Eiffel frees the programmer from this burden, thanks to a garbage collector.

It was once widely believed that garbage collection was quite expensive relative to explicit heap management, but recent advances in garbage collection technology make automatic storage reclamation affordable for use in high-performance systems. Generational techniques reduce the basic costs and disruptiveness of collection by exploiting the empirically-observed tendency of objects to die young (objects from the same generation can be allocated and collected in a row [20]). Incremental techniques may even make garbage collection relatively attractive for real-time systems [12]. Most Eiffel implementations come with such an incremental garbage collector, which can be activated and suspended nearly at will.

In the case of the implementation of our SMDS server, we measured that the garbage collector's work is shorter when it is called often, and that it globally needs less than 1% of the computation time. So, with frequent iterative collections, its work duration has a Gaussian kind of distribution with an average around 3 ms. As a result, our SMDS server launches an iterative collection each time it is otherwise idle (not to penalize packet processing), but if a working period is too long, a collection is forced so the optimal interval between two collections is respected, and the collecting time never exceeds 15 ms and has a probability of 99.9% to be under 10ms. While this approach practically meets the requirements expressed in Section 2.3.1, it is clear that things would have been easier if a truly real-time

Kind of processing	headers processed/s	mean processing time
SNI to ISSI	5025	199 us
ISSI to SNI	8093	124 us
Switching (ISSI to ISSI)	5347	187 us

Table 1. Header processing speed in an SMDS server

garbage collector would have been available in our development environment.

Further that allowing us to limit the garbage collector monopolization of the processor to short periods, the shifting of memory management processing to idle periods allows time savings during active periods. Our server then has a higher ability to absorb traffic peaks.

4 Implementation and Validation Process

4.1 Qualitative Validation

The output of the design stage gave us a class hierarchy, allowing us to build an actual prototype for each class (with stub routines), and then implement the routine bodies class by class. After a class is actually implemented, it is tested separately to validate its internal consistency (unit testing). It is then added to the system *in lieu* of its stub class, and the system is tested against this new set of functionalities (see [8] for more details).

Testing takes place in the framework of *Design by Contract* [16, 11], that prompts designers to specify precisely every consistency condition that could go wrong, and to assign explicitly the responsibility of its enforcement to either the routine caller (the client) or the routine implementation (the contractor). Assertions (taking the form of class invariants, preconditions, and postconditions) are the mechanisms that enable the formalization of this contract between a client and a contractor. By enabling the proper compile-time switch, these assertions can be monitored at runtime, which provides a powerful help in the testing process.

Basically, a party failing to meet the contract terms indicates the presence of a fault. A postcondition violation points out a bug in a routine implementation, which does not fulfill its promises. On the other hand, a precondition violation points out a contract broken by the client: this is very useful while in (incremental) integration testing, because the already integrated code can be protected from mishandling by the newly integrated one.

The problem is that evaluating assertions at run time costs processing time, which can be disturbing in a real-time setting. We solved this problem in an ad hoc fashion by substituting a logical clock to the real one during functional testing: real time issues were to be tested separately, as described below.

4.2 Quantitative Validation

To get real performance figures from our parallel SMDS server, we implemented it on the Intel Paragon XP/S supercomputer, a distributed-memory multicomputer with architecture that can accommodate more than a thousand heterogeneous nodes connected in a two-dimensional rectangular mesh —its computation nodes are based on Intel i860 processors, and communicate by passing messages over a high-speed internal interconnect network.

Our implementation made use of our own Eiffel Parallel Execution Environment (EPEE) [7], that can be seen as a kind of a toolbox. It mainly consists of a set of cross-compilation tools that allow the generation of executable code for many different distributed computers (Intel Paragon XP/S, Fujitsu AP1000, SGI, network of Unix workstations, etc.). It also includes a set of Eiffel classes that provide facilities for sharing data across a distributed computer and for handling data exchanges between its processors in a portable way.

4.2.1 Header processing speed

We first determine the internal performance limits of our SMDS server. The significant figures are the speed of header processing in different contexts: transmission (packet received from an SNI and then injected in the SMDS network), reception (PDU coming from an ISSI link and delivered to a SNI), and switching of traffic (from an ISSI link to an other one).

For these specific measures, we use a specialized SMDS server, where the lower layers are simulated: a transmission only consists in incrementing a counter; and as for receptions, the server is always told that a PDU is ready to be read and the reception is simulated (a fixed set of predefined PDU is used). The measures consist in performing continual operations on the server and compute their mean durations (which is more realistic than exploring the assembly language listing to add up individual times of machine language instructions on a given path of the header processing).

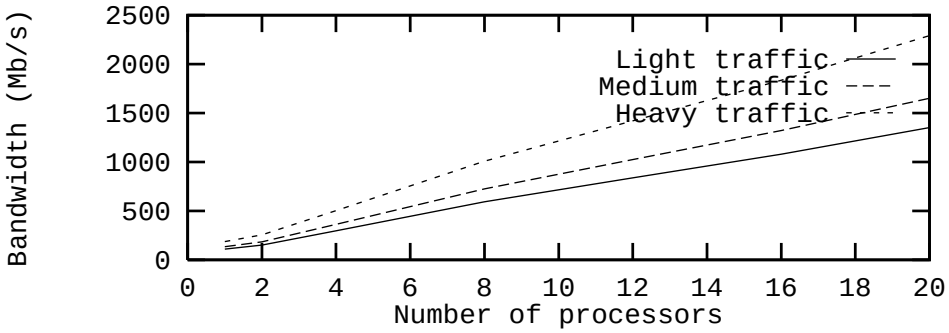

Figure 4. Maximal aggregate bandwidth in optimal conditions

Since these tests involve no network connection, the results are directly proportional to the processing power of the processor. The performance figures exposed in Table 1 show that the internal speed of a sequential server is sufficient to reach Gigabit flow rates on a standard processor: the slower operation reaches 3.29 Gb/s with 64k PDU, thus easily meeting the requirements of Section 2.3.2.

4.2.2 Measuring Aggregate Bandwidth

These performance tests consist in measuring the maximal switching capacity of a parallel server, depending on the number of processor it has. The test architecture includes:

- a parallel server implemented on 1, 2, 4, 8, 12, 16 and 20 nodes of a Paragon XP/S
- an environment (surrounding the parallel server) made of a number of other SMDS servers (e.g., 24), each one implemented on one node of the Paragon XP/S and achieving traffic generation and absorption.

We are interested in seeing how faster does a parallel server work, depending on the number of nodes it has. So we measure data flow rates for various packet sizes, under different conditions of (random) traffic load.

From all these measures (various packet sizes and traffic profiles), we extract the best aggregates bandwidth results for each size of SMDS servers (from 1 to 20 nodes, for 3 kinds of traffic densities). These results are displayed on Figure 4.

A single processor server achieves user data switching at a speed of 180Mb/s, whereas a 20 processors parallel servers reach nearly 2.3Gb/s, widely above the Gigabit data flow rate given as an unlikely reachable limit in [13]. We obtain a quasi-linear speed-up, with an average efficiency of 66%. Extrapolating this result, even greater flow rates should be achievable by increasing the number of nodes of the parallel SMDS server.

5 Conclusion

We have outlined the design and the implementation of a scalable SMDS server using a fully object-oriented approach, from analysis to design to implementation. We have shown that the use of advanced object-oriented technologies (such as high modularity, dynamic binding, design by contract, automatic garbage collection, etc.) are not incompatible with soft real-time computing. Still, a good optimizing compiler is mandatory to remove the bulk of the late binding calls and bypass the overhead due to OO fine-grain modularity to produce small and efficient object code. Another *sin equa non* condition is that the OO runtime environment must make it possible to control the garbage collector behavior, e.g., by providing hooks to activate and suspend its activity at will.

In our SMDS case study, we were able to build an implementation featuring high-throughput and low-delay transmissions and respecting all the real-time constraints of SMDS.

Using a high level programming environment presents a number of advantages. First, since the aggregate bandwidth of the server is proportional to the number of supporting nodes, it makes easily available a *range* of performances adjustable to the user needs (because no new software has to be written: since in Eiffel everything is dynamic, the software can configure itself at boot-time, using e.g., an *Abstract Factory*). Then, since most of the software is fully portable (only the code dealing with device drivers is system dependent), it can easily be tested on the development station where the actual device drivers are only simulated. Easier portability also means we can benefit from the exponential increasing of processing power to produce servers that run twice as fast as the previous generation for free.

Acknowledgments

We would like to thank Xavier Desmaison, Frederic Guerber and Isabelle Levern who made it possible to transform the ideas presented in this paper into actual software. We also would like to thank Michel Train and Remi Houdaille (Lucent Technologies, Rennes) for their comments on early versions of this paper.

References

[1] P. America. Pool-T: A parallel object-oriented programming. In A. Yonezawa, editor, *Object-Oriented Concurrent Programming*, pages 199–220. The MIT Press, 1987.

[2] Bellcore. Generic requirements for smds networking. Technical Report TA-TSV-001059, Bell Communication Research, 1992.

[3] G. Berry and A. Benveniste. The synchronous approach to reactive and real-time systems. *Another Look at Real Time Programming, Proceedings of the IEEE*, 79:1270–1282, 1991.

[4] T. Braun and M. Zitterbart. Parallel XTP implementation on transputers. In *The 1991 Singapore International Conference on Networks*, pages 91–96. G.S.Poo, Sep 1991.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.

[6] F. Guidec, J.-M. Jézéquel, and J.-L. Pacherie. An object oriented framework for supercomputing. *Journal of Systems and Software*, Special Issue on *Software Engineering for Distributed Computing*, June 1996.

[7] J.-M. Jézéquel. EPEE: an Eiffel environment to program distributed memory parallel computers. *Journal of Object Oriented Programming*, 6(2):48–54, May 1993.

[8] J.-M. Jézéquel. *Object Oriented Software Engineering with Eiffel*. Addison-Wesley, Mar. 1996. ISBN 1-201-63381-7.

[9] J.-M. Jézéquel. Reifying configuration management for object-oriented software. In *International Conference on Software Engineering, ICSE'20, Kyoto, Japan*, Apr. 1998.

[10] J.-M. Jézéquel, X. Desmaison, and F. Guerber. Performance issues in implementing a portable SMDS server. In IFIP, editor, *6th International IFIP Conference On High Performance Networking*, pages 267–278. Chapman & Hall, London, Sept. 1995.

[11] J.-M. Jézéquel and B. Meyer. Design by contract: The lessons of Ariane. *Computer*, 30(1):129–130, Jan. 1997.

[12] E. K. Kolodner and W. E. Weihl. Atomic incremental garbage collection. In *Proc. Int. Workshop on Memory Management*, number 637 in Lecture Notes in Computer Science, pages 365–387, Saint-Malo (France), September 1992. Springer-Verlag.

[13] J.-Y. Le Boudec, A. Meier, R. Oechsle, and H. L. Truong. Connectionless data service in an atm-based customer premises network. *Computer Networks and ISDN Systems*, 0(26):1409–1424, July 1994.

[14] S. Matsuoka and A. Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object Oriented Programming*. MIT Press, 1993.

[15] J. Meseguer. Solving the inheritance anomaly in concurrent object-oriented programming. In O. Nierstrasz, editor, *Proceedings ECOOP'93*, LNCS 707, pages 220–246, Kaiserslautern, Germany, July 1993. Springer-Verlag.

[16] B. Meyer. Applying "design by contract". *IEEE Computer (Special Issue on Inheritance & Classification)*, 25(10):40–52, Oct. 1992.

[17] B. Meyer. *Eiffel: The Language*. Prentice-Hall, 1992.

[18] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, New Jersey, 1991.

[19] D. C. Schmidt. Reactor: An object behavioral pattern for concurrent event demultiplexing and event handler dispatching. In *Pattern Languages of Program Design*, volume 1. Addison-Wesley, 1995.

[20] R. Sharma and M. L. Soffa. Parallel generational garbage collection. In *Proceedings OOPSLA'91*, pages 16–32, Nov. 1991. Published as ACM SIGPLAN Notices, volume 26, number 11.

[21] P. Stephan. Building financial software with object technology. *Object Magazine*, 5(4), July 1995.

[22] A. Tantawy. Réalisation de protocoles à haute performance. In *Actes du colloque CFIP'93 sur l'ingénièrie des protocoles, Montreal*. Hermès, Sept. 1993.

[23] A. Yonezawa, J.-P. Briot, and E. Shibayama. Object-oriented concurrent programming in ABCL/1. In *OOPSLA'86 Proceedings*, September 1986.

[24] O. Zendra, D. Colnet, and S. Collin. Efficient dynamic dispatch without virtual function tables: The SmallEiffel compiler. In *Proceedings OOPSLA '97, ACM SIGPLAN Notices*, Oct. 1997.

[25] M. Zitterbart. High-speed transport components. *IEEE Network Magazine*, pages 54–63, Jan. 1991.