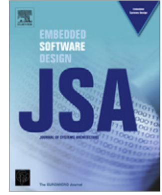
Journal of Systems Architecture 54 (2008) 868–876

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc



Securing skeletal systems with limited performance penalty: The `muskel` experience ☆

Marco Aldinucci, Marco Danelutto *

Computer Science Department, University of Pisa, Largo B. Pontecorvo 3, I-56127 Pisa, Italy

ARTICLE INFO

Article history:
Received 11 July 2007
Received in revised form 16 November 2007
Accepted 10 January 2008
Available online 5 March 2008

Keywords:
Skeletons
Parallelism
Security
Scalability

ABSTRACT

Algorithmic skeletons have been exploited to implement several parallel programming environments, targeting workstation clusters as well as workstation networks and computational grids. When targeting non-dedicated clusters, workstation networks and grids, security has to be taken adequately into account in order to guarantee both code and data confidentiality and integrity. However, introducing security is usually an expensive activity, both in terms of the effort required to managed security mechanisms and in terms of the time spent performing security related activities at run time.

We discuss the cost of security introduction as well as how some features typical of skeleton technology can be exploited to improve the efficiency code and data securing in a typical skeleton based parallel programming environment and we evaluate the performance cost of security mechanisms implemented exploiting state of the art tools. In particular, we take into account the cost of security introduction in `muskel`, a Java based skeletal system exploiting macro data flow implementation technology. We consider the adoption of mechanisms that allow securing all the communications involving remote, unreliable nodes and we evaluate the cost of such mechanisms. Also, we consider the implications on the computational grains needed to scale secure and insecure skeletal computations.

© 2008 Elsevier B.V. All rights reserved.

1. Introduction

Cole introduced algorithmic skeletons in late 1980 [2]. An algorithmic skeleton is nothing but a known, parametric parallelism exploitation pattern. It can be customized by programmers providing suitable parameters in such a way as to match the needs of the particular application at hand. Usually, skeletons can also be nested in such a way that users/programmers can express very complex parallelism patterns as composite skeletons. As originally intended by Cole, algorithmic skeletons represent a good trade-off between expressive power and efficiency in the field of parallel/distributed programming.

Typical examples of skeletons are task farms (parallel computation of the same function on a set of completely independent input tasks. This kind of computation is often referred to as "embarrassingly parallel" computation), pipelines (parallel computation of stages of a computation on different input task items), map, reduce and parallel prefix (parallel computation of the same function on all the elements of a data structure, "summation" of all the elements of a data structure using a binary, associative and commutative function, and same kind of "summation" computing both final value and all intermediate sum values) and several flavors of iterator skeletons (modeling different loop schemata).

Algorithmic skeletons led to the development of several *skeletal systems*, that is parallel programming environments exploiting the skeleton concept in different flavors: libraries, new languages, coordination languages and patterns. Examples of such programming frameworks include both programming languages and libraries. In the former case, new languages have been designed that include skeletons as language primitives/constructs. In the latter case, skeletons are supported by proper library calls hosted in plain, existing sequential languages such as C/C++ or Java. Examples of skeleton programming languages are P3L [3] and ASSIST [4,5]. These are both programming languages designed and implemented by our group in Pisa in 1991 and in 2000, respectively. Examples of skeleton libraries are eSkel [6–8], Muesli [9], Skipper [10] and `muskel` [11]. eSkel and Muesli are implemented in C and C++, respectively, and they both use MPI [12] to exploit parallelism in skeleton computations. They have been recently designed by Cole and Kuchen, respectively. Skipper is implemented in Ocaml instead; it runs on top of plain TCP/IP workstation networks and uses the same macro data flow implementation model of `muskel`. Finally, `muskel` is our pure Java/RMI skeleton library derived from Lithium [13] and it is the library we used to perform the experiments discussed in this paper.

☆ Expanded version of [1]. This work has been partially supported by FP6 Network of Excellence CoreGRID funded by the European Commission (Contract IST-2002-004265).

* Corresponding author.
E-mail addresses: aldinuc@di.unipi.it (M. Aldinucci), marcod@di.unipi.it (M. Danelutto).

1383-7621/$ - see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.sysarc.2008.02.008

Most of these systems have been designed and implemented to target workstation clusters and they happen to be very efficient when dedicated, homogeneous clusters are taken into account. However, when more typical heterogeneous, non-dedicated workstation networks and grids are considered, several additional implementation problems have to be dealt with: on the one hand, firewalls and high network latencies have to be taken into account; on the other hand security issues have to be safely handled.

Security issues, in particular, arise when parallel programs are executed on distributed architectures whose remote nodes and/or clusters are interconnected by means of public, non-dedicated network infrastructures. In this case both code (the one staged to remote nodes for the execution) and data (input data and computation results) flow to and from remote nodes through potentially insecure network links. Data and code crossing insecure links can be easily snooped or spoofed by persons that are not those actually performing the parallel computation. The solution to this problem is therefore to secure all the connections flowing through non-secure network links. State-of-the-art, de facto standard tools can be used to the purpose. As an example, SSL sockets can be used to connect nodes through insecure network links. However, this operation might have a non-negligible performance penalty. This cost is to be paid both on sending and receiving machines: the sending machines should spent time to encrypt serialized data during the marshaling process and the receiving machine should spent time to decrypt incoming data in the unmarshaling procedures. This is pure overhead from the functional application viewpoint, that is, the effort spent securing communications is not directly related to the computation of the functional result of the parallel program. In addition, further time has to be spent while transmitting the encrypted data through network links due to the fact that encrypted data is usually longer than original data. As a consequence, programming environments succeeding to secure only those communications that turn out to be "sensitive" (e.g. those involving secret code/data *and* flowing through shared, public network links) will perform much better than environments exploiting different security policies. In particular, they will perform better than those environments indiscriminately securing all the communications taking place during the parallel application execution, as these environments will demonstrate a larger communication overhead. Also, they perform better than those environments not supporting secure communications at all, as these environments will not guarantee user data and code confidentiality.

In the case of classical parallel programming environments, the task of identifying the sensitive code/data parts and the subsequent task of implementing suitable and efficient security mechanisms to protect sensitive code/data is completely in the charge of the application programmers. The proper usage of security tools is not easy, however. The usage of even slightly different security tools may lead to very different performance and will provide very different security guarantees. As an example, consider the adoption of different encrypting suites/algorithms in the usage of SSL sockets in Java. The encrypting algorithms can be changed playing with the parameters passed to SSL socket factories, usually. Different algorithms provide different confidentiality degrees and may involve significantly different encrypting/decrypting times. Therefore application programmers must have a fairly good knowledge of the available security tools to exploit them decently in the application code.

Currently available skeletal systems do not support any kind of security feature. The MPI libraries by Cole and Kuchen are intended to be run on MPI clusters, that usually exploit private, secure networks. Therefore, attention has been concentrated on other features related to efficiency and expressive power. ASSIST was designed to run on grids, either exploiting the Globus toolkit [14] or using plain TCP/IP POSIX workstation mechanisms. In the latter case, it uses `ssh` and `scp` to perform remote commanding and data and code staging to and from remote machines. However, we never measured the impact of the usage of the `ssh`/`scp` tools. Recently, the Muenster university group leaded by Gorlatch introduced HOC [15,16]. High order components (HOC) is a grid-programming environment jointly exploiting skeleton technology and component technology. HOC includes predefined components providing programmers with pipeline and task farm parallelism exploitation patterns. The implementation uses web services to manage grid related issues, such as data and code staging. At the moment, however, security issues are not yet taken into account in HOC although there are specifications to put security over standard XML/SOAP protocols used in web services [17].

More attention is paid to security issues in non-skeleton based grid programming system. For instance, the Globus grid middleware [14] provides a full range of tools to handle security issues [18]. Overall, security is one of the key points to be addressed according to the NGG reports [19]. Recently, in the framework of the CoreGRID European Network of Excellence [20], security has been considered an "horizontal issue" that is an issue to be considered in all the Institutes of the network, and a useful survey of security grid related issues has been produced [21]. We considered the results of all these experiences before investigating the impact of security in skeletal systems.

In this work, we try to figure out the order of the costs in securing communications in a skeletal system and then we show how proper security strategies can be adopted that do not necessarily involve the application programmer in the security policies implementation process. At this aim, we used `muskel` as testbed skeletal system. Section 2 introduces `muskel`. Section 3 outlines the security related issues and discusses how they can be addressed in the `muskel` skeletal system. Section 4 presents and discusses some experimental results achieved with the secure `muskel` system. Finally, Section 5 discusses how aspect oriented programming (AOP) techniques can be used to support security in skeletal systems.

2. The `muskel` Java skeleton library

`muskel` is a full Java, skeleton based, parallel programming library [11,22]. It can be used to run parallel skeleton programs on clusters or networks of workstations as well as computational grids. The only requisite of `muskel` is that the distributed processing nodes all support standard Java (1.5 or later) and RMI and that they can be accessed via standard `ssh`/`scp` tools. `muskel` provides the user with a set of fully nestable stream parallel skeletons (pipelines and farms). Skeletons are implemented by transforming the user supplied skeleton program into a data flow graph. Then, each task to be computed is used to provide the input token to a copy of such graph, which is placed in a logically centralized graph pool. The fireable instructions[1] in the graph pool are then scheduled for execution onto remote data flow interpreter nodes. The result tokens computed at the remote nodes are either used as input tokens to different macro data flow instruction in the graph pool (possibly making such instructions fireable) or to be output as the results of the program execution.

The whole process is completely transparent to the user (the application programmer). The users only have to provide code such as the sample one depicted in Fig. 1. In this case, we assumed that two Java classes exist that process medical images coming from some kind of scanner (PET, CAT, MNR) to filter (class `Filter`) them and then to suitably render (class `Render`) the filtered images. The stream of images to be processed is taken from a file by properly exploiting the `boolean hasNext()` and `Object next()` methods

[1] A data flow instruction becomes "fireable" when all its input tokens are available.

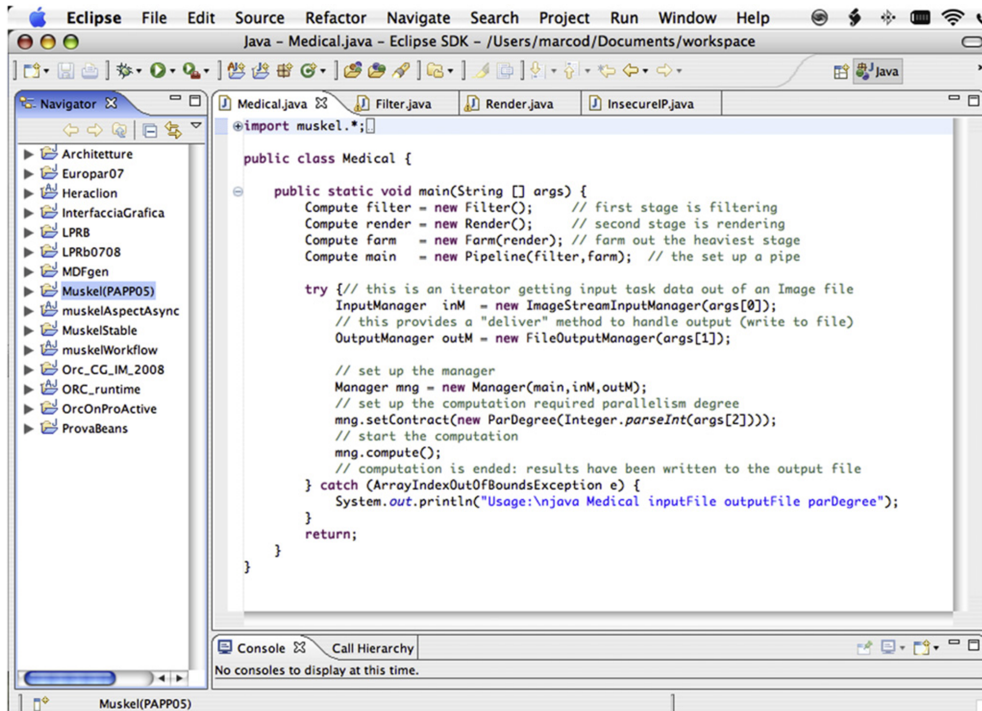


Fig. 1. Sample `muskel` code.

implemented in the user defined input manager class. The result images will eventually be stored in another file, invoking the `void deliver(Object r)` method implemented by the user defined output manager class. The user asks to compute the program using as many remote data flow interpreter nodes as indicated by the command line argument passed to the program (the program is run with the standard Java interpreter on the user machine by issuing a command such as `java Medical filein.dat fileout.dat 10`). Furthermore, as he knows the rendering phase takes significantly longer than the filtering one, he asks to execute in parallel the second stage of this pipeline computation, by writing the second stage of the pipeline as a farm.

`muskel` uses `Manager`s to manage computations. The manager takes a skeleton program, an input and an output manager, and a performance contract (the parallelism degree, in this case). Then it arranges to discover and recruit a suitable number of remote interpreter nodes and forks a `ControlThread` for each of the recruited interpreters. The process of recruiting remote interpreters can be executed in two different ways. In one case (version 1.0 of `muskel`), the manager retrieves the addresses of the remote machines from a text file hosting a ⟨machinename, port⟩ pair list. In another case (current version of `muskel`, 2.0), a peer-to-peer discovery protocol (exploiting UDP multicast) is started that eventually gathers answers from the remote machines where an interpreter was running hosting the same ⟨machinename, port⟩ info.

Each `ControlThread` forked by the manager executes a loop. At each loop iteration:

(i) It fetches a fireable instruction from the macro data flow (MDF) graph pool ((①) in Fig. 2).
(ii) Delivers the fireable instruction to the remote interpreter (②).
(iii) Gets back the results of the remote computation (③).
(iv) Eventually either it stores the results (④) as proper tokens in the MDF graph pool or, if they are final results, it delivers them to the output manager.

We assume here the "logically centralized" macro data flow graph pool is actually implemented through centralized, data structures in the user JVM address space (the current implementation of `muskel` actually implements the graph pool as a macro data flow instruction `Vector` declared in the `Manager`; the reasoning

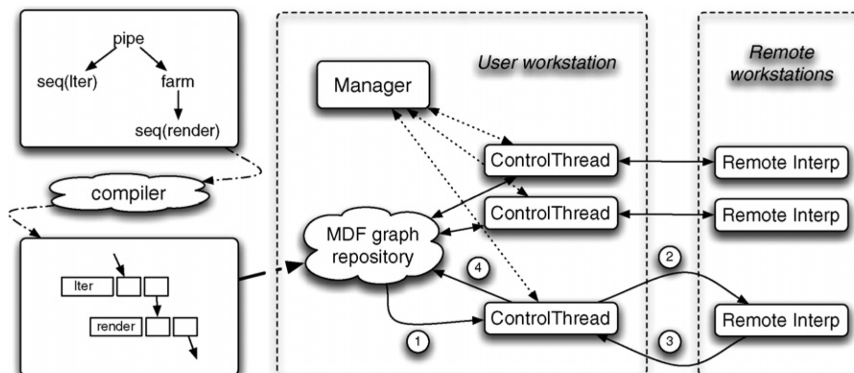


Fig. 2. `muskel` functioning.

and the results discussed in the following also hold in the case of distributed implementation of the graph pool). As a consequence, all the communications taking place during execution of a generic muskel program happen to be either communication of program (chunk) code to the remote macro data flow interpreters or communication of input or output tokens to or from the remote interpreter instances.

The muskel manager also arranges to instantiate a fresh copy of the MDF graph in the MDF graph repository for each of the tasks retrieved using the input manager, with the task placed as a token in the appropriate MDF "initial" instruction of the graph. If there is a problem with one of the remote interpreters (a remote node fault or a network problem) the ControlThread informs the manager and terminates. In turn, the manager tries to recover the situation by recruiting a new remote interpreter and putting back the uncomputed fireable instruction in the MDF graph repository. Overall, this mechanism satisfies and maintains the user supplied performance contract (ParDegree) without any user/programmer intervention, implementing de facto an autonomic self-configuration policy [23].

The remote interpreter instances are launched on the remote nodes, possibly using a shell script once and for all (remote interpreters are plain Java remote objects running as standalone processes or as Java Activatable objects). They are specialized to execute the code of the application at hand by the ControlThreads forked by the manager. The ControlThreads deliver to the interpreters the serialized version of the relevant Compute classes once and for all just before starting the delivery of fireable MDF instructions.

muskel has been tested on several configurations of networked workstations including plain, dedicated clusters, local network of different, heterogeneous production workstations and geographical scale networks hosting different machines in two sites separated by firewalls.[2] In all the cases, almost perfect scalability has been achieved, provided that suitably coarse grain programs are run. We showed that local network configurations (i.e. configurations hosting processing elements in a single LAN) scale well with skeleton code involving computations with a grain (as defined in Section 4) of the order of 10. Geographical scale networks, instead, required computations with a significantly larger grain (one to two order of magnitude larger than the one scaling on the local network).

3. Experimenting security in muskel

When exploiting parallelism using nodes that are interconnected by public network links there is always the risk that communications are intercepted and relevant data is snooped by unauthorized people. Also, data can be snooped and substituted with other wrong or misleading data exploiting spoofing techniques, thus leading to incorrect computations. An even worst case concerns code. Consider what happens in muskel: serialized code is sent to the remote interpreters that is then used to compute remotely the fireable macro data flow (MDF) instructions related to the user skeleton code. If such code is changed, the remote nodes can be used to compute things they were not supposed to compute. Therefore, it is fundamental, in order to avoid both data and code problems, that

(i) the access to the remote interpreters is authenticated in a secure way, and
(ii) that the code itself is encrypted before being sent to the remote interpreters.

[2] ProActive [24] was used in this case to perform RMI call tunneling through ssh.

Authentication and code encrypting can be easily programmed using Java JSSE extensions, included in the JDK since version 1.4. Consequently, we modified the muskel prototype to provide authentication, data confidentiality and integrity in the communications taking place among the control threads running on the user machine and the remote data flow interpreter instances running on the remote machines. In particular, we prepared a muskel version exploiting the Java SSL library to perform communications involving remote processing nodes. SSL provides authentication exploiting asymmetric keys, and data confidentiality and integrity exploiting a symmetric session key and message digests. Overall, SSL represents a well known and assessed tool to secure remote communications over TCP. We then used the modified version of muskel (we will refer to it as *secure* muskel from now on) to evaluate the impact of security on the raw performance of the skeletal system. Just to avoid interferences or difficulties in evaluating the experimental results due to any kind of additional mechanism, we stripped down the current muskel prototype by replacing the RMI remote interpreter access with plain TCP/IP sockets connections. In secure muskel we used the very same code modified only in the parts opening the sockets. Those parts dealing with the opening of plain TCP/IP sockets were modified to host the opening of SSL connections through proper calls to the SSL socket factories provided by Java 1.5. This process resulted in the implementation of two distinct versions of the base muskel engine able to compute in a distributed/parallel way sets of macro data flow instructions stored in the fireable instruction pool. The two versions have been used to evaluate the costs related to the introduction of security in the skeletal system, through the experiments described in the following section.

4. Experimental results

In order to figure out how the introduction of secure remote communications affect the execution of muskel programs we performed a set of experiments. All the experiments have been run on a Fast Ethernet network of Pentium III machines running Linux with a vendor modified 2.4.22 kernel (Figs. 3–5). Java networking experiments have been run on the platform mentioned above (Fig. 6, left), and on a Giga Ethernet network of AMD dual-core Opteron275 machines running Linux with 2.6.9SMP kernel (Fig. 6, right).

4.1. Secure muskel vs. plain muskel

The first set of experiments measured the performance achieved when running the same muskel skeleton program a workstation network first using the original muskel prototype, with insecure communications, and then using the secure muskel prototype. We considered programs with different *computational grain*, i.e. programs whose macro data flow instructions have a different average computation to communication time ratio. In other words, we first defined computational grain G as

$$G = \frac{T_w}{T_c}$$

Here T_w represents the time spent by a remote interpreter instance to compute the macro data flow instruction on the local data and T_c represents the time spent in transferring the input data to the remote interpreter instance plus the time spent getting back the computed results from the remote interpreter instance. Then we measured the performance of several programs with different values of G. Fig. 3 shows the results we achieved running the same experiments with the non-secure (left) and secure (right) muskel system. In the legend, $W = x/C = y$ means that the average T_w of

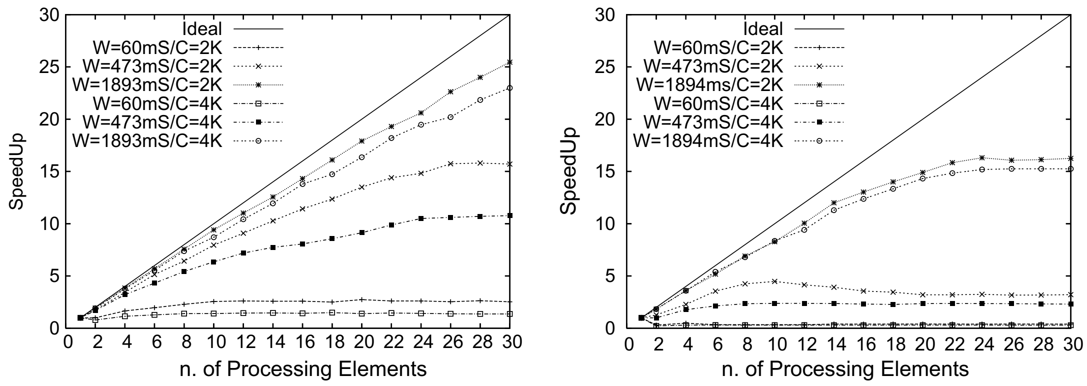


Fig. 3. (left) `muskel` speedup (plain TCP/IP sockets); (right) secure `muskel` speedup (SSL).

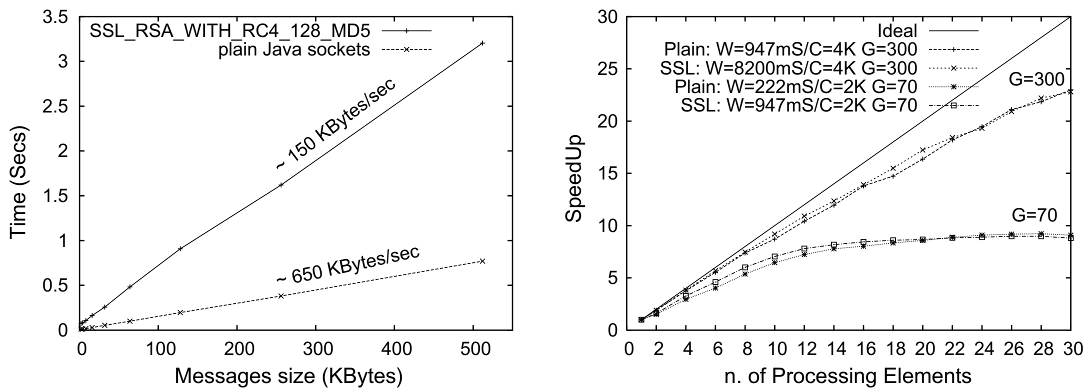


Fig. 4. (left) `muskel` vs. secure `muskel` bandwidth (serialization time is included); (right) effect of grain on speedup.

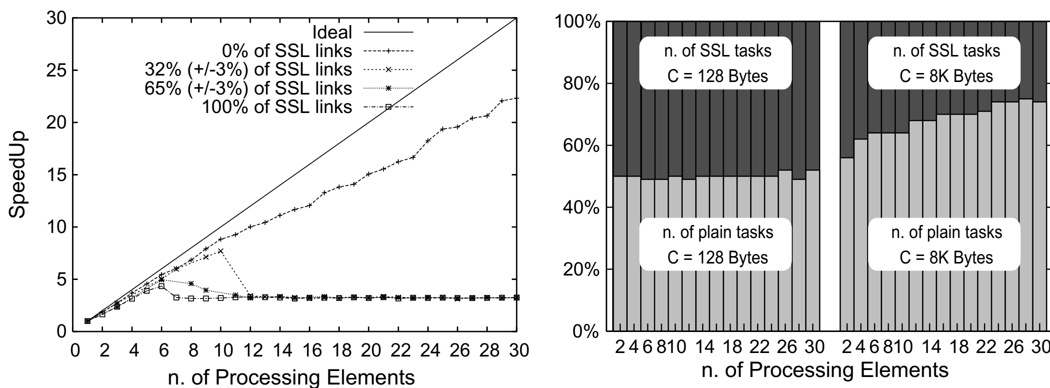


Fig. 5. `muskel` with selectively introduced SSL communication links. (left) Speedup vs. percentage of SSL links; (right) distribution of tasks among workers interconnected through plain and SSL links for 128 and 8 kbytes tasks.

macro data flow instructions was x and the average amount of input data transferred to the remote interpreter to compute the instruction plus the amount of output data retrieved from the remote interpreter was y bytes. The workstations used were dedicated to `muskel` runs, the programs were the same, the input data were also the same and therefore the only factor influencing the completion times is the usage of the SSL sockets. Both plots, the `muskel` and the secure `muskel` ones are actual speedup plots, rather than scalability plots: the point corresponding to one processing element describes the sequential execution of the macro data flow instructions on a single processor, rather than the usage of just one remote interpreter instance. In this case, no communication overhead at all is counted in the execution time.

4.2. SSL vs. plain TCP/IP bandwidth

We measured the raw communication bandwidth of `muskel` and secure `muskel` in order to be able to correctly interpret the results of our experiments. Fig. 4 (left) shows the bandwidth achieved in the two cases. The lower bandwidth of secure `muskel` is mostly due to the overhead introduced at processor level due to the encrypting/decrypting activity taking place at the sending and

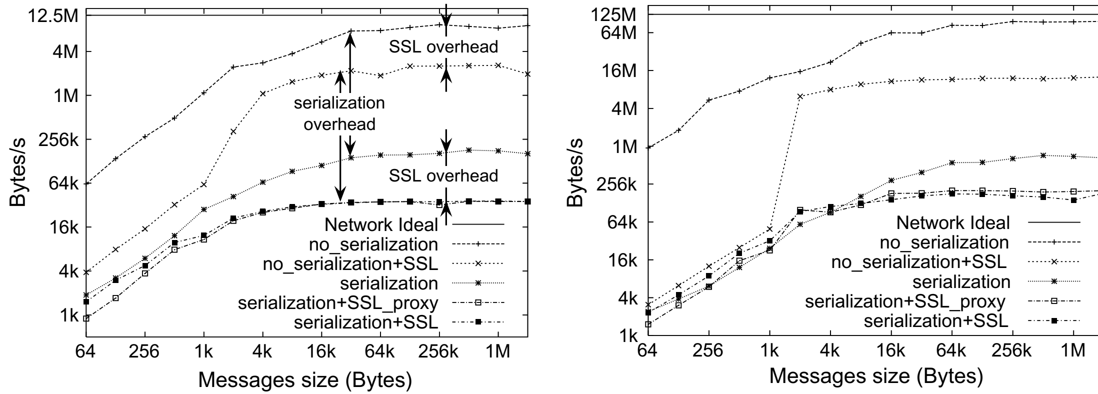


Fig. 6. Java socket performances with raw messages (`byte []`), serialized messages (`Integer []`), raw messages on SSL (TLS_DHE_DSS_WITH_AES_128_CBC_SHA), serialized messages on SSL, serialized messages through an SSL proxy thread on two different clusters: (left) PIII@800 MHz 100 Mbit/s Ethernet, and (right) dual-core Opteron275@2.2 GHz 1 Gbit/s Ethernet.

receiving node. It is only partially due to the initial key exchange handshake, which is performed once and for all, and to the slightly longer (about 10%, actually) message encoding used in SSL. From these measures, we can conclude that the introduction of SSL affects the computational grain G needed to mask the longer communication times involved in `muskel` computations. As a consequence, we expected that coarser grain programs (that is programs with higher values of G) are needed to achieve good secure `muskel` performance figures.

4.3. Iso-grain speedup

In order to evaluate the effect of computational grain on speedup, we ran another experiment. We chose different values of G and ran programs with that G value on both `muskel` and secure `muskel` prototype. As the T_c values depend on the communication library, we had to use larger data flow instructions in the secure `muskel` runs to get the same G with the same amount of data transferred to and from the remote interpreter instances. Fig. 4(right) shows the results achieved in this experiment. When the grain is high $G = 300$, both `muskel` and secure `muskel` scale pretty well (also in this case, the plot is related to speedup, not to scalability). However, the secure `muskel` run required computations significantly longer than the standard `muskel` run in order to achieve comparable speedups. Actually, secure `muskel` required computations eight times longer than those required by standard `muskel` to reach the $G = 300$ value that led to *quasi*-linear speedup. This is due to the overhead involved to the usage of SSL sockets. When computational grain is smaller, however, both `muskel` and secure `muskel` stop scaling quite early, as shown by the $G = 70$ plots in the same figure.

4.4. Skeleton related optimizations

The experimental results clearly show that the costs involved in security handling are definitely not negligible. Although this is not a "brand new" nor an unexpected result, the numbers in the plots give a precise dimension to the cost of introducing security. Being related to skeleton based parallel programming, they also show how the impact is relevant despite the relative simplicity of the run time support used. The clear and simple structure of the `muskel` run time, in fact, makes evident that the overhead measured is coming only from the (correct) usage of the SSL support.

It is clear, then, that we must figure out how such costs can be optimized. In particular, we must be able to exploit the knowledge available at compile and run time, derived from the analysis of the structure of both the skeleton program and of the process network used to implement it, to improve the efficiency of the secure version of `muskel`. One kind of knowledge, we can exploit in this process is the knowledge related to the location of the remote processing elements recruited to act as remote MDF interpreter nodes. The `muskel` manager arranges to recruit remote MDF interpreter nodes either using a peer-to-peer discovery service or consulting some kind of machine list configuration file. At the end of the recruitment process, the IP addresses of these machines are known. By comparing these addresses with the address of the workstation the user is currently using to run the program, the manager can figure out, immediately before starting the control threads managing the remote MDF interpreter nodes, which ones are local (that is belong to the same LAN as the user machine running the `muskel main`) and which ones are not. Presumably, the local nodes happen to operate in a controlled environment, and therefore they can be reached with plain TCP/IP instead than using a more costly secure communication mechanism. Non-local nodes, on the other hand, *must* be reached by using a secure mechanism if the network path to the nodes flows through public networks or generically insecure links. With this information available, the `muskel Manager` can thus decide whether to fork a plain control thread (the one using plain TCP/IP RMI) or a secure control thread (the one using SSL RMI) for each of the remote processing elements recruited with a very small amount of additional code.

We therefore ran another experiment: we modified secure `muskel` to use SSL only with non-local nodes and to use plain TCP/IP sockets with the local nodes. Then, we ran the same program on two clusters, with the same kind of machines, that is Linux machines with the same processors and the same amount of memory. One cluster was in the same network as the user machine running the `main muskel` program. The other cluster was remote and therefore was managed by SSL `muskel` control threads. Actually, to remove the problem in the result analysis deriving from the different latencies in reaching local and remote nodes, we configured part of the local nodes as if they were non-local. Therefore, again, the only difference was in the usage of SSL `muskel` control threads rather than plain, non-SSL control threads. The results are shown in Fig. 5. Fig. 5(left) shows the speedups achieved in runs of the same program performed using a variable mix of the distributed data flow interpreter instances placed on local machines and on remote machines. The speedups achieved in the mix runs are clearly smaller than the ones reached in runs only involving the local/secure nodes. However, the `muskel` manager and control threads implement a self-adapting load balancing strategy. Each control thread only dispatches a new fireable MDF instruction when the results

of the execution of the previous one have been received. Therefore "slow" remote interpreters get fewer tasks to be computed compared to "fast" ones. Fig. 5(right) shows the measured percentage of fireable instructions (tasks) computed by each one of the remote interpreters. In case the amount of data transferred to the remote interpreter instance is small (left part of the figure), and therefore the weight of encrypt/decrypt is small, local and remote instances get about the same amount of tasks to be computed. However, when the amount of data transferred becomes significant (right part of the figure), the remote interpreter instances get fewer tasks to be computed, due to the combined effect of the longer time spent in communications and of the load balancing mechanism. Actually, this control mechanism was thought to solve load balancing in case of usage of heterogeneous workstations (different CPUs, different amounts of central store or even different operating systems) but it proved very effective also in this case.

4.5. Serialization and encrypting overhead

As introduced in Section 2, the `muskel` manager forks several `ControlThreads` to distribute MDF instructions to remote MDF interpreters. In particular, fireable MDF instructions are actually (code, data) pairs. The easiest and the more Java orthodox way to distribute this kind of objects involves the Java native serialization mechanism, which is a very general but quite inefficient process. The `muskel` version used in the reported experiments is compliant to this view. However, it has been optimized to avoid the full serialization of all fireable MDF instructions. In particular, the complete MDF graph is serialized and uploaded just once to all remote interpreters, while the fireable MDF instructions actually sent to the remote interpreters are codified to refer to the nodes of this graph. On the other hand, the data component of MDF instructions can be serialized with a lightweight, *ad-hoc* marshaling procedure, although this does not happen in the current version of the `muskel` prototype.

Experiments in Fig. 6 break down the two main overhead sources of the `muskel` implementation with respect to MDF instruction distribution: serialization and encrypting. Abstractly, both of them are aspects of MDF communication that can be independently considered. As is clear from Fig. 6, both Java serialization cost and SSL securing significantly affect communication performance. Despite the fact that the exact balance between the two overheads can be determined by considering only the particular serialization and encrypting processes (e.g. data type, algorithm, key length), experiments highlight that the serialization process is in general heavier (more CPU demanding) than encrypting.

4.6. Network impact

The results discussed in the previous sections (but those of Fig. 6, right) must be considered while taking into account that the network used for the experiments was Fast Ethernet. The NIC (network interface cards) used do not support any kind of on-board data processing. In particular, the cards used just provide hardware support to access via DMA the packets to and from the main store from and to the internal buffer. Therefore, the whole cost of securing messages, i.e. the whole cost of encrypting data packets, is paid by the central CPU serially to the time spent by the NIC to actually send the message. In other words, the classical cost model for communication that assumes a cost of

$$T_{\mathrm{comm}}(\#\mathrm{bytes}) = t_{\mathrm{init}} + \#\mathrm{bytes} \times t_{\mathrm{byte}}$$

where t_{init} represents the cost of preparing the message and initializing the NIC and t_{byte} is basically the inverse of the network bandwidth has to be read considering that t_{init} incorporates not only data encapsulation in proper packets of the protocol stack but also the encrypting of the payload using the symmetric key negotiated in the SSL setup phase.

The additional messages exchanged in the initial phase of the establishment of a SSL connection to negotiate the symmetric session key, on the other hand, are paid just once and for all, as the connections between remote MDF interpreters and the management control threads are established once and for all when the manager is asked to start the `muskel` parallel code execution with the `manager.compute()` method call. Therefore these additional messages (with respect to plain TCP/IP connection setup) add a negligible overhead in the case that non-trivial (long) input streams are processed.

If different network hardware is used, things may change a lot. In particular, if modern, high performance network hardware is used, such as QsNet *II* [25], then NIC on board processors can be exploited to implement payload encryption. In this case, the overhead on the CPU falls back to the same class as the overhead paid in the case of plain TCP/IP usage. According to this scenario, we introduced in each communication channel a pair of encrypting proxies. They can be placed at deployment time between two communicating partners *A*, and *B* to secure their communications via SSL. A proxy is realized as a thread that inter-operates with *A* on the one end (via Java shallow memory copy), and with *B* (or its proxy) on the other end (via a SSL-secured socket). The thread runs the encrypting algorithm on all messages exchanged with *B*. As shown in Fig. 6, the SSL proxy introduction has little or no impact on communication performance with respect to standard SSL communications. In addition, in the case of dual-core machine (Fig. 6, right), the thread may run on a different core with respect to the communication partner, thus improving the overall communication performance. It is worth pointing out that the benefit of the approach cannot be fully sensed with just a communication experiment, such as the one leading to the results of Fig. 6. In this case, the communications performed are "rendez-vous" communications: the single `ControlThread` asks for the evaluation of a fireable MDF instruction and then waits for its results. In addition to a slight improvement of absolute communication performance, the SSL proxies free the partners from encrypting CPU work that is moved to another thread that may run on another core. This may result in a significant improvement of communication performance for asynchronous partner interactions, such as stream-oriented interactions. In particular, such stream data traffic is the one involved by the usage of a suitable proxy on the user machine taking care of all the communications happening between `ControlThreads` and remote interpreter instances. As sketched in Fig. 7, the approach can be naturally generalized to consider also data serialization. We believe these techniques will naturally match the current trend of CPU architecture, which is moving towards an ever increasing number of cores per CPU, by using some of CPU resources as communication-oriented co-processors.

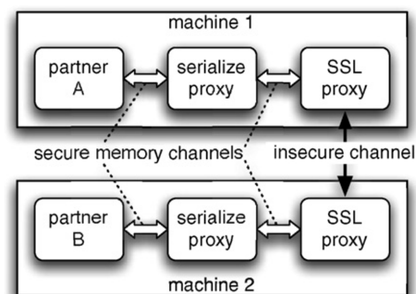


Fig. 7. Encrypting and serializing proxies in a pipeline.

We are currently experimenting the autonomic usage (as self-protection mechanism) of encrypting proxies between components of a distributed application for grid [26]: each component of a distributed application can autonomously and dynamically negotiate with wired partners the introduction of encrypting proxies to deal with insecure links.

5. Exploiting skeletons to drive security handling

As discussed above, the introduction of security in distributed systems might have significant performance drawbacks. In skeleton systems these drawbacks can be ameliorated by selectively introducing secured channels just along communication paths that are considered insecure. This technique is particularly effective on a macro data flow implementation of skeleton systems because of the implicit load balancing these systems guarantee among macro data flow interpreters.

In this section, we discuss how the `muskel` skeleton programming model can be exploited to provide the user convenient tools to drive security introduction in the `muskel` interpreter. In particular, we discuss how the user can tell the `muskel` system which is the sensitive code and data, as well as communicating hints about the "untrusted" nodes, i.e. those nodes that *always* need secured communications, independently of their network location and connections. This kind of information is then exploited in the run time of the skeleton system to selectively introduce security in the places where it is actually needed.

In order to allow users to provide `muskel` interpreter such information, we exploit Java annotations and aspect oriented programming (AOP), that is we use state-of-the-art Java metaprogramming tools. Programmers may use annotations to distinguish both sensitive code and sensitive data. To denote sensitive code, users may use the `@SensitiveCode` marker annotation, whereas sensitive data can be denoted with a `@SensitiveData` annotation. Both annotations should be used to tag `Compute` object `compute` methods. As an example, considering the code in Fig. 1, we can assume that all the patient data are to be considered sensitive, whereas only the `Render` code is sensitive. Therefore, we can annotate the code as follows:

```
public class Filter extends Compute {
   @SensitiveData(''INOUT'')
   public Object compute(Object task) {
      Image inImg = (Image) task;
      ...
      return filteredImg;
   }
}
```

```
public class Render extends Compute {
   @SensitiveCode
   @SensitiveData(''INOUT'')
   public Object compute(Object task) {
      Image inImg = (Image) task;
      ...
      return renderedImg;
   }
}
```

meaning that the computation of the image rendering is code that must be kept confidential when the code is staged to remote macro data flow interpreters connected through public, shared network links, and that both input parameters and output results have to be kept confidential both when computing `Filter` and `Render`.

Users can also provide insecure node IP information using the `InsecureIP` annotation to annotate the manager declaration:

```
@InsecureIP(''alpha1'',''131.114.2.14'')
Manager mng = new Manager(main,inM,outM);
```

The information provided by the user is *qualitative* information. No detail is to be given by the programmers about the mechanisms to be used to implement security policies. The whole task of taking care of the security issues is delegated to the `muskel` run time. Exploiting AOP techniques and introspection, calls to the remote nodes related to the evaluation of `@SensitiveCode` annotated `compute` method are processed in such a way that the corresponding code is staged to the remote data flow interpreter through an SSL connection. Also, data representing input tokens related to a `@SensitiveData(''IN'') compute` methods are sent to a remote interpreter over SSL connections and `@SensitiveData(''OUT'')` results (output tokens) are retrieved from the remote workers using SSL connections rather than plain, unsecured TCP/IP sockets. Finally, the manager can introspect the IP addresses known to be insecure, according to the user/programmer hints, and arrange to setup `SecureControlThread`s instead of plain `ControlThreads`[3] to manage those remote nodes.

Overall, this approach allows the clear separation between the concerns about qualitative security aspects handling and the concerns related to the actual implementation of the strategies to implement security aspects in the execution of the skeleton program, and in particular

(i) It is in line with the skeleton programming model (programmers only give hints to the skeleton system about the way parallelism should be exploited, not details about *how* parallelism exploitation is to be implemented).
(ii) It implements the separation of concerns inherent in the AOP methodology.
(iii) It delegates to the `muskel` run-time the choice of the more suitable mechanisms to implement the security policies "suggested" by the user through the annotations, leaving the possibility to the `muskel` run-time to exploit all the knowledge available related to the execution environment and to the target architecture.

6. Conclusions

We discussed the cost of introducing security features into a skeletal system such as `muskel`, i.e. a skeletal system implemented exploiting macro data flow technology. We modified the `muskel` implementation in such a way that the communications taking place between the remote machines are performed guaranteeing authentication, data confidentiality and integrity, by exploiting the SSL Java library. We evaluated the cost of such operation. Then we showed how the exploitation of the information available at run time can mitigate its high cost.

We discussed experiments that clearly confirm that indiscriminate adoption of security techniques has a high cost and that exploiting proper information (either provided by the user or gathered from the execution environment) in the structured programming environment may mitigate the overhead induced by security

[3] `SecureControlThread` performs the same tasks as a `ControlThread1` but uses secured communications to handle remote macro data flow interpreter calls rather than plain TCP/IP connections.

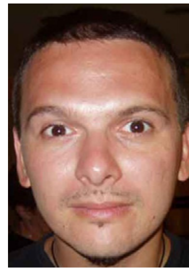
mechanisms. As security is a fundamental issue in highly distributed systems, such as multi-cluster and grid architectures, we think this could be considered an interesting contribution to the skeletal system implementation technology.

Acknowledgement

We are grateful to Peter Kilpatrick for his substantial help in improving the presentation of this work.

References

[1] M. Aldinucci, M. Danelutto, The cost of security in skeletal systems, in: P. D'Ambra, M.R. Guarracino (Eds.), Proceedings of the International Euromicro PDP 2007: Parallel Distributed and Network-based Processing, IEEE, Napoli, Italia, 2007, pp. 213–220.
[2] M. Cole, Algorithmic skeletons: structured management of parallel computations, Research Monographs in Parallel and Distributed Computing, Pitman, 1989.
[3] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, M. Vanneschi, P^{3}L: a structured high level programming language and its structured support, Concurrency Practice and Experience 7 (3) (1995) 225–255.
[4] M. Vanneschi, The programming model of ASSIST, an environment for parallel and distributed portable applications, Parallel Computing 28 (12) (2002) 1709–1732.
[5] M. Aldinucci, A. Petrocelli, E. Pistoletti, M. Torquati, M. Vanneschi, L. Veraldi, C. Zoccolo, Dynamic reconfiguration of grid-aware applications in ASSIST, in: J.C. Cunha, P.D. Medeiros (Eds.), Proceedings of 11th International Euro-Par 2005 Parallel Processing, vol. 3648 of LNCS, Springer, 2005, pp. 771–781.
[6] M. Cole, Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming, Parallel Computing 30 (3) (2004) 389–406.
[7] A. Benoit, M. Cole, S. Gilmore, J. Hillston, Flexible skeletal programming with eSkel, in: J.C. Cunha, P.D. Medeiros (Eds.), Proceedings of the 11th International Euro-Par 2005 Parallel Processing, vol. 3648 of LNCS, Springer, Lisboa, Portugal, 2005, pp. 761–770.
[8] A. Benoit, M. Cole, S. Gilmore, J. Hillston, Using eskel to implement the multiple baseline stereo application, in: G.R. Joubert, W.E. Nagel, F.J. Peters, O. Plata, P. Tirado, E. Zapata (Eds.), Parallel Computing: Current & Future Issues of High-End Computing, Proceedings of the PARCO 2005, vol. 33 of NIC, Research Centre Jülich, Germany, 2005, pp. 673–680.
[9] H. Kuchen, A skeleton library, in: B. Monien, R. Feldman (Eds.), Proceedings of the Eighth International Euro-Par 2002 Parallel Processing, vol. 2400 of LNCS, Springer, Paderborn, Germany, 2002, pp. 620–629.
[10] J. Serot, D. Ginhac, Skeletons for parallel image processing: an overview of the SKiPPER project, Parallel Computing 28 (12) (2002) 1685–1708.
[11] M. Danelutto, QoS in parallel programming through application managers, in: Proceedings of the International Euromicro PDP: Parallel Distributed and Network-based Processing, IEEE, Lugano, Switzerland, 2005, pp. 282–289.
[12] MPI forum home page, 2007. <http://www.mpi-forum.org/>.
[13] M. Aldinucci, M. Danelutto, P. Teti, An advanced environment supporting structured parallel programming in Java, Future Generation Computer Systems 19 (5) (2003) 611–626.
[14] Globus web site, 2007. <http://www.globus.org/>.
[15] J. Dünnweber, S. Gorlatch, HOC-SA: a grid service architecture for higher-order components, in: IEEE International Conference on Services Computing, Shanghai, China, IEEE, 2004, pp. 288–294.
[16] M. Alt, J. Dünnweber, J. Müller, S. Gorlatch, HOCs: higher-order components for grids, in: Component Models and Systems for Grid Applications, CoreGRID, Springer, 2005, pp. 157–166.
[17] C. Geuer-Pollmann, J. Claessens, Web services and web service security standards, Information Security Technical Report 10 (1) (2005) 15–24.
[18] V. Welch, F. Siebenlist, I. Foster, J. Bresnahan, K. Czajkowski, J. Gawor, C. Kesselman, S. Meder, L. Pearlman, S. Tuecke, Security for grid services, in: Proceedings of the 12th International Symposium on High Performance Distributed Computing (HPDC), IEEE, Los Alamitos, CA, USA, 2003, pp. 48–57.
[19] Next Generation GRIDs Expert Group, NGG3, Future for European Grids: GRIDs and Service Oriented Knowledge Utilities, Vision and Research Directions 2010 and Beyond, 2006. <ftp://ftp.cordis.lu/pub/ist/docs/grids/ngg3_eg_final>.
[20] CoreGRID web site, 2007. <http://www.coregrid. net/>.
[21] CoreGRID NoE deliverable series, Institute on Trust and Security, Deliverable D.IA.03 – Survey Material on Trust and Security, 2005. Available at: http://www.coregrid.net/.
[22] M. Danelutto, P. Dazzi, Joint structured/non-structured parallelism exploitation through data flow, in: V. Alexandrov, D. van Albada, P.M.A. Sloot, J. Dongarra (Eds.), Proceedings of the ICCS: International Conference on Computational Science, Workshop on Practical Aspects of High-level Parallel Programming, LNCS, Springer, Reading, UK, 2006.
[23] J.O. Kephart, D.M. Chess, The vision of autonomic computing, IEEE Computer 36 (1) (2003) 41–50.
[24] ProActive home page, 2007. <http://www-sop.inria.fr/oasis/proactive/>.
[25] J. Beecroft, D. Addison, D. Hewson, M. McLaren, D. Roweth, F. Petrini, J. Nieplocha, QsNet^{II}: defining high-performance network design, IEEE Micro 25 (4) (2005) 34–47.
[26] M. Aldinucci, S. Campa, M. Danelutto, P. Dazzi, P. Kilpatrick, D. Laforenza, N. Tonellotto, Behavioural skeletons for component autonomic management on grids, in: CoreGRID Workshop on Grid Programming Model, Grid and P2P Systems Architecture, Grid Systems, Tools and Environments, Heraklion, Creete, Greece, 2007.



Marco Aldinucci got the PhD in Computer Science in 2003, he has been researcher at the Institute of Information Science and Technologies of the Italian National Research Council (ISTI/CNR, 2003–2006), and he is currently research associate at Computer Science Department of the University of Pisa, Italy. He is author of more than 50 papers appearing in journals and international refereed conference proceedings, together with more than 40 different co-authors. He has been and is currently participating in more than 10 national and international research projects concerning parallel computing, autonomic computing, and Grid topics, including the Grid.it Italian National Project, CoreGRID EC Network of Excellence, GridComp EC-STREP, BEinGRID EC-IP, INSYEME MIUR-FIRB. His main research is focused on parallel/distributed computing in network of workstations and grids, and in particular on models and tools for high-level parallel programming, autonomic computing, component-based frameworks and distributed shared memory systems. He led the design and the development of a number of tools for parallel processing, including compilers, libraries and frameworks, both in industrial and academic teams.

Marco Danelutto is an Associate Professor at the Department of Computer Science, University of Pisa, since 1998. His main research interests are in structured parallel/distributed/grid programming (algorithmic skeletons, parallel design patterns, macro-data flow implementation models), autonomic computing, software components and semi formal methods and tools supporting parallel and distributed computing. He is currently leading the Programming model Institute within CoreGRID EU Network of Excellence and he has been formerly the leader of the GRID.it (Italian national FIRB project) workpackage responsible of the ASSIST programming environment design and implementation. In the past, he has been one of the main designers of P3L and he is currently maintaining Muskel, a full Java skeleton library targeting generic networks of Java enabled workstations. Marco Danelutto is author and co-author of about 110 papers appearing in international journals and refereed conferences.