The cost of security in skeletal systems

M. Aldinucci
Dept. Computer Science
Univ. of Pisa – Italy
aldinuc@di.unipi.it

M. Danelutto
Dept. Computer Science
Univ. of Pisa – Italy
marcod@di.unipi.it

Abstract

Skeletal systems exploit algorithmical skeletons technology to provide the user very high level, efficient parallel programming environments. They have been recently demonstrated to be suitable for highly distributed architectures, such as workstation clusters, networks and grids. However, when using skeletal system for grid programming care must be taken to secure data and code transfers across non-dedicated, non-secure network links. In this work 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 taking place between remote, unreliable nodes and we evaluate the cost of such mechanisms. In particular, we consider the implications on the computational grains needed to scale secure and insecure skeletal computations.*

Keywords: *skeletons, parallelism, security, scalability.*

1. Introduction

Algorithmical skeletons represent a good tradeoff between expressive power and efficiency in the field of parallel/distributed programming. An algorithmical 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 by nesting simple skeletons users/programmers can exploit very complex parallelism patterns. Typical examples of skeletons are task farms (modeling embarrassingly parallel computations), pipelines (modeling computations organized in stages), map, reduce and prefixes (modeling classical apply-to-all and sum-up data parallel computations) and several flavors of iterator skeletons (modeling different loop schemas).

After being introduced by Cole [8], algorithmical 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 implementing skeleton programming languages are P3L [4] and ASSIST [20, 2]. They are both programming languages designed and implemented by our group in Pisa in '91 and in 2000 respectively. eSkel [9, 6], Muesli [16], Skipper [18] and `muskel` [11] are examples of libraries providing parallel skeletons. The first two are implemented in C and C++ and run on top of MPI. They have been recently designed by Cole and Kuchen respectively. Skipper is implemented in Ocaml instead, runs on top of plain TCP/IP workstation networks and uses the same macro data flow implementation model of `muskel`. `muskel` is our pure Java/RMI skeleton library derived from Lithium [1] and it is the library we used to perform the experiments discussed in this paper.

Recently, we developed two skeleton based programmnig environments, ASSIST and `muskel`, both targeting heterogeneous workstation networks and grids. With these environments, several further implementation problems have to be dealt with: on the one side, firewall and high network latencies have to be taken into account, and on the other side security issues have to be safely handled.

In particular, security issues arise when skeleton programs are executed on distributed architectures whose remote nodes and clusters are interconnected via public and/or 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) are flowing 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 managing to perform the parallel computation. The answer is then to secure the connections flowing through non-secure network links. But this has a cost that has to be

[0]This research work is carried out under the FP6 Network of Excellence CoreGRID funded by the European Commission (Contract IST-2002-004265).

IEEE COMPUTER SOCIETY

paid both on sending and receiving machines (that must cipher and decipher (possibly serialized) code and data) and in terms of network bandwidth (ciphered connections may require more communications and/or differently sized messages to complete).

In this paper, we try to figure out the order of the costs in securing communications in a skeletal system. The skeletal system used is `muskel`, which we shortly describe in Section 2. Section 3 outlines the security related issues and how they can be addressed in the `muskel` skeletal system. Eventually, Sec. 4 will present and discuss some experimental results achieved with secure `muskel` system.

2. muskel

`muskel` is a full Java, skeleton based, parallel programming library [11, 12]. It can be used to run parallel skeleton programs on network of workstations that support Java and RMI. It 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. The fireable instructions in the graph are then scheduled for execution onto remote data flow interpreter nodes, and the result tokens computed are either used to fire new instructions or to be output as the results of the program execution. All this process is completely transparent to the user that only has to provide code such as the one in Figure 1. Here 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 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 10 remote data flow interpreter nodes. Furthermore, as he knows the rendering phase takes sensibly 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 `Managers` to manage computations. The manager takes a skeleton program, input and output managers 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 control thread for each one of the recruited interpreters. The control thread enters a loop. In the loop body, it fetches a fireable instruction from the MDF (Macro Data Flow) graph repository ①, delivers it to the remote interpreter ②, gets the results of the remote computation ③ and eventually either delivers the results ④ as tokens in the MDF graph or, in case they are final results, it delivers them to the output manager. The manager also arranges to instantiate a fresh copy of the MDF graph in the MDF graph repository for each one of the tasks retrieved using the input manager, with the task placed as a token in the proper MDF "initial" instruction of the graph. In case there is a problem with one of the remote interpreters (a remote node fault or a network problem) the control thread 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.

The interpreters 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 control threads forked by the manager. The control threads deliver to the interpreters the serialized version of the relevant `Compute` classes just before starting the delivery of fireable MDF instructions. The process of recruiting remote interpreters can be executed in two different ways. In one case (version 1.0 of `muskel`), the addresses of the remote machines are retrieved by the manager 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[1] is started that eventually gathers answers from the remote machines were an interpreter was running hosting the same ⟨machinename, port⟩ info.

`muskel` has been tested on several configuration of networked workstations including plain, dedicated clusters (RLX Pentium III Blade chassis hosting 24 nodes), local network of different, production workstations (Pentium III to IV running Linux and Mac OS X Power PC G4 and G5 machines interconnected via Fast Ethernet and several Linux and Mac OS X laptops connected via wireless), geographical scale network hosting the same kind of 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 (ratio between the time spent in computing a remote DF instruction and the time spent in delivering the input tokens and retrieving the output tokens) around 100. Geographical scale networks, instead, required computations with a sensibly larger grain (1 to 2 orders bigger than the one scaling on the local network).

[1] the discovery service exploits UDP multicast

[2] ProActive [17] was used in this case to perform RMI call tunneling through `ssh`

IEEE COMPUTER SOCIETY

```
import muskel.*;

public class SampleCode {

 public static void main(String [] args) {

   // first of all define the program to be computed
  Compute filter = new Filter();                                          // first stage is filtering
  Compute render = new Render();                                         // second stage is rendering
  Compute farm = new Farm(render);     // as rendering is heavier than filtering, let's farm it out in the pipe
  Compute main = new Pipeline(filter, farm);          // this is the program we eventually compute

   // then arrange to provide input "task" stream and some way to manage results
  InputManager inM =      // provide input image stream (implement hasNext/next abstraction to get images)
       new ImageStreamInputManager("sample_mage.dat");
  OutputManager outM =          // store results (provide deliver method to handle each result computed)
      new FileOutputManager("sample_result.dat");

   // now arrange to declare a manager: it will completely take care of the parallel computation
  Manager mng = new Manager(main,inM,outM);                                   // declare the manager
  mng.setContract(new ParDegree(Integer.parseInt(args[0])));  // ask to use args[0] remote PEs

   // now ask to compute the program
  mng.compute();                                // start the computation and wait for termination

   // that's all: results are now in the sample_result.dat file ...
 }
}
```

Figure 1. Sample `muskel` code

3. Introducing security

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 foreign, 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. Take into account 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 relative 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 1) the access to the remote interpreters is authenticated in a secure way and 2) that the code itself is ciphered before being sent to the remote interpreters.

Authentication and code ciphering can be easily programmed using Java JSSE extensions, included in the JDK since version 1.4. Therefore, we decided to modify the `muskel` prototype to provide authentication, privacy 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 Java SSL library to perform communications involving remote processing nodes. SSL provides authentication, privacy, and integrity. These features are provided through asymmetric keys, symmetric session keys and message digests, respectively. Overall, SSL represents a well known and assessed tool to secure remote communications over TCP. We then used the modified version of `muskel` (we'll 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 just 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

15th EUROMICRO International Conference on Parallel, Distributed and Network-Based Processing (PDP'07)
0-7695-2784-1/07 $20.00 © 2007 IEEE
IEEE COMPUTER SOCIETY

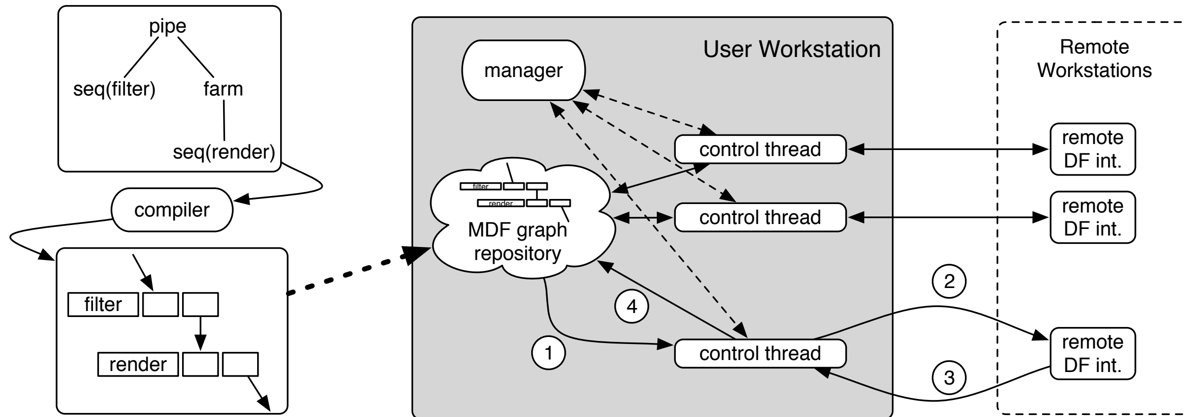


Figure 2. `muskel` **functioning**

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. Experiments with secure `muskel`

In order to figure out how the introduction of secure remote communications impact on the execution of `muskel` programs we performed a set of experiments. All the experiments were run on Fast Ethernet networks of Pentium III machines running Linux with a vendor modified 2.4.22 kernel.

4.1. Secure `muskel` vs. plain `muskel`

The first set of experiments measured the performance achieved when running the same `muskel` skeleton program onto 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 defined computational grain G as

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

where 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, and then we measured the performance of several programs with different values of G. Figure 3 shows the results we achieved. The left plot is relative to the original `muskel` runs and the right one is relative to the runs using the secure `muskel` version. In the legend, $W = x/C = yK$ means that the average T_w of macro data flow instructions was x and the 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 Kbytes. The workstations used were dedicated to `muskel` runs, the programs were the same, the input data were the same also 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 relative to 1 processing element is relative to the sequential execution of the macro data flow instructions on a single processor, rather than to 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. Figure 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 ciphering/deciphering activity taking place at the sending and 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 this measures we can conclude that the introduction of SSL impacts on 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`

IEEE COMPUTER SOCIETY

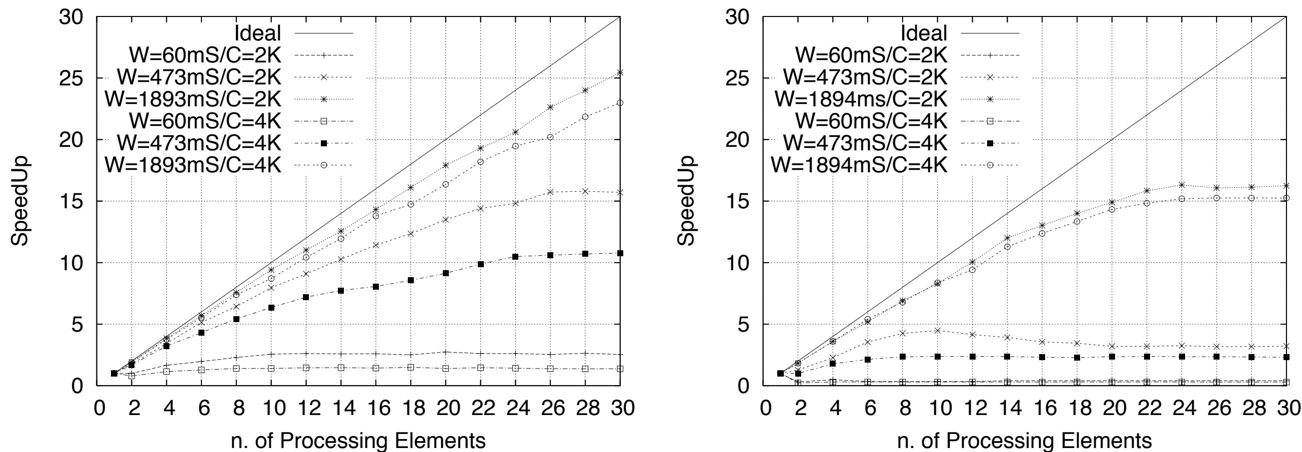


Figure 3. SSL vs. plain TCP/IP socket

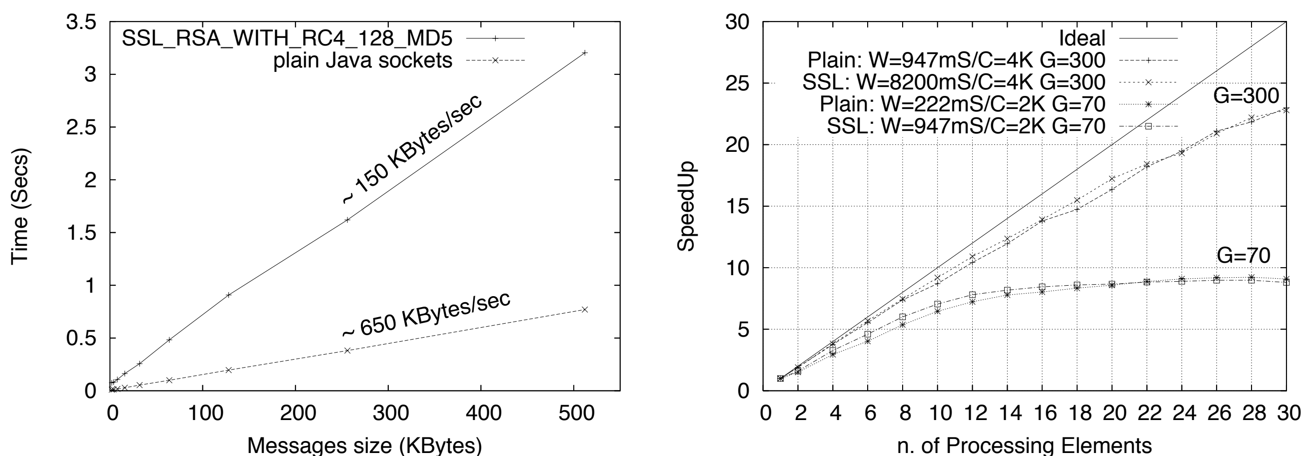


Figure 4. `muskel` **vs secure** `muskel` **bandwidth (left, times include serialization time) and effect of grain (right)**

performance figures.

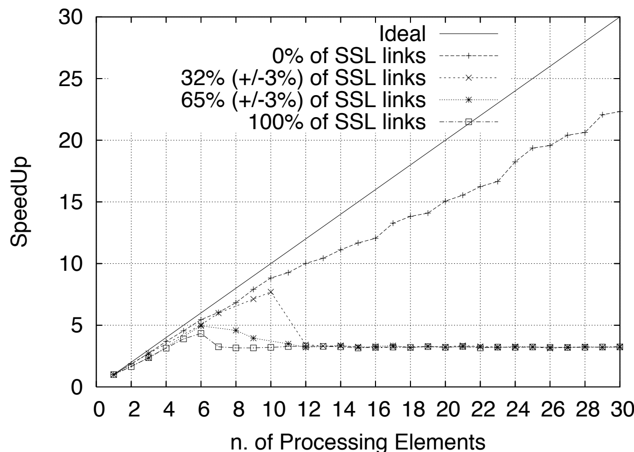
4.3. Iso-grain speedup

In order to evaluate the effect of computational grain on speedup, we run another experiment. We choose different values of G and run 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. Figure 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 relative 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 8 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 pretty 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 secure coding are definitely not negligible. Al-

15th EUROMICRO International Conference on Parallel, Distributed and Network-Based Processing (PDP'07)
0-7695-2784-1/07 $20.00 © 2007 IEEE
IEEE COMPUTER SOCIETY



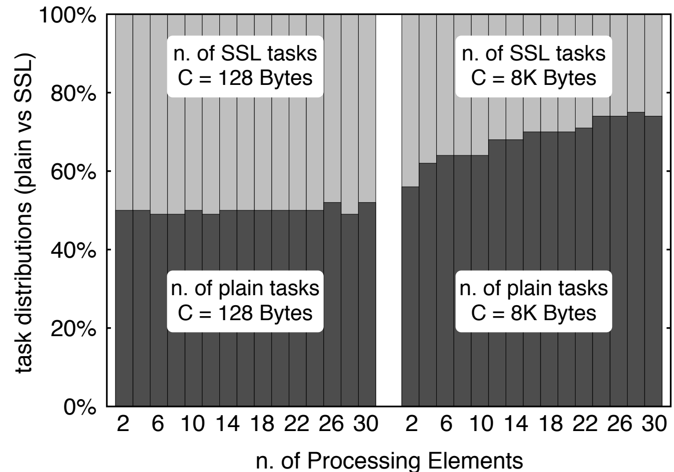


Figure 5. Mixed local nodes/non SSL and remote nodes/SSL `muskel` execution

though 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 only coming 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 relative 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 peer2peer discovery service or consulting some kind of machine list configuration file. At the end of the recruit process, the IP addressese 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 of the user machine running the `muskel main`) and which ones are not. Presumably, the local nodes happen to operate in controlled environment, and therefore they can be reached with plain TCP/IP instead than using more costly secure communication mechanism. Non-local nodes, on the other hand *must* be reached exploiting 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 one of the remote processing elements recruited with a very small amount of additional code.

We therefore run 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 run 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 central memory. One cluster was in the same network of 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 Figure 5. Figure 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 one reached in the local/insecure nodes only runs. However, `muskel` manager and control thread 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 with respect to "fast" ones. Figure 5 right shows the measured percentage of fireable instructions (tasks) computed by each one of the remote interpreters. In case the amount

IEEE COMPUTER SOCIETY

of data transferred to the remote interpreter instance is small (left part of the Figure), and therefore the weight of cipher/decipher is small, local and remote instances get more or less 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 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 demonstrated very effective also in this case.

Although we have no experimental results on the this topic yet, we just outline another source of optimizations related to security. When writing structured parallel programs according to the skeleton model, the *structure* of the parallel program is clearly exposed by the programmer himself. It is quite to introduce suitable *annotations* that can be used by the programmer to denote which parameters are "sensible", that is which are the communications that must be eventually secured. As an example, we may suppose to provide two marker annotations `@secureData` and `@secureCode` to denote those `compute` methods that *do use* sensible parameters or, respectively, sensible code, and therefore should exploit the secure communication framework ther in the transmission of fireable data flow instructions to the remote interpreters or in the transmission of the code to the remote interpreters during the initialization phase of the distributed macro data flow interpreter. By properly exploiting these annotations, we could combine the analysis on secure and insecure nodes outlined above with the knowledge supplied by the programmer. Eventually, we'll be able to secure just those communications that involve sensible data and/or code *and* use non secure communication links. Again, this shows how the knowledge typically available in a skeleton system can be usefully exploited to reduce the impact of security to the minimum actually necessary.

4.5. Network impact

The results discussed in the previous sections must be considered 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, that means the whole cost of ciphering 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

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

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} comprehends not only data encapsulation in proper packets of the protocol stack but also the cyphering of the payload using the symmetric key negotiated in the SSL set up 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 exceution with the `manager.compute()` method call. Therefore these additional messages (with respect to plain TCP/IP connection set up) add a negligible overhead in case non trivial (long) input streams are processed.

In case different network hardware was used, things may change a lot. In particular, if modern, high performance network hardware was used, such as QsNet*II* [5], then NIC on board processors can be exploited to implement payload ciphering. In this case, the overhead on the CPU falls back to the same class of the overhead paid in case of plain TCP/IP usage. Still, we expect the optimization introduced by exploiting the information derived from the skeleton structure of the program improves the overall efficiency of `muskel` programs, as avoiding unnecessary ciphering of message payload frees NIC coprocessor resources that can be consequently exploited for different purposes.

5. Related work

Currently available skeletal systems do not support any kind of security feature. The MPI libraries by Cole and Kuchen are thought to be run on MPI clusters, that are usually exploiting private, secure networks. Therefore the 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 [15] or exploiting plain TCP/IP POSIX workstation mechanisms. In this latter case, it actually 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 [3, 13]. HOC (High Order Components) is a grid-programming environment jointly exploiting the skeleton technology and component technology. HOC provides predefined components providing the 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 [7].

More attention is paid to security issues in non-skeleton based grid programming system. The globus grid middleware [15] provides a full range of tools to handle security issues [21], for instance. And security is one of the key points to be addressed accordingly to the NGG reports [19]. Recently, in the framework of the CoreGRID European Network of Excellence [10], security has been considered an "horizontal issue" that is an issue to be considered in all the Institutes of the network, and a nice survey of security grid related issues has been produced [14]. We considered the results of all this experiences before investigating the impact of security in skeletal systems.

6. Conclusions

We discussed the cost of introducing security features into a skeletal system. The skeletal system taken into account is `muskel`, our all Java skeleton based parallel programming library. We modified the implementation in such a way as to ensure the communications taking place between the remote machines with authentication, privacy and integrity, by exploiting SSL. 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. The experiments discussed are not claimed to be definitive nor complete. We are currently planning and performing further experiments aimed at demonstrating that secure skeletal programming environments can be designed that also achieve decent performance results, exploiting these first results we achieved with the secure `muskel` prototype. 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.

References

[1] M. Aldinucci, M. Danelutto, and P. Teti. An advanced environment supporting structured parallel programming in Java. *Future Generation Computer Systems*, 19(5):611–626, 2003. Elsevier Science.

[2] M. Aldinucci, A. Petrocelli, E. Pistoletti, M. Torquati, M. Vanneschi, L. Veraldi, and C. Zoccolo. Dynamic reconfiguration of grid-aware applications in ASSIST. In *11th Intl Euro-Par: Parallel and Distributed Computing*, volume 3648 of *LNCS*, pages 771–781, Lisboa, Portugal, August 2005. Springer Verlag.

[3] M. Alt, J. Dünnweber, J. Müller, and S. Gorlatch. HOCs: Higher-order components for grids. In V. Getov and T. Kielmann, editors, *Component Models and Systems for Grid Applications*, CoreGRID, pages 157–166. Springer Verlag, June 2004.

[4] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi. P^3L: A Structured High level programming language and its structured support. *Conc. Practice and Experience*, 7(3):225–255, 1995.

[5] J. Beecroft, D. Addison, D. Hewson, M. McLaren, D. Roweth, F. Petrini, and J. Nieplocha. $QsNet^{II}$: Defining High-Performance Network Design. *IEEE Micro*, 25(4):34–47, January-February 2005.

[6] A. Benoit, M. Cole, S. Gilmore, and J. Hillston. Flexible Skeletal Programming with eSkel. In *11th Intl Euro-Par: Parallel and Distributed Computing*, volume 3648 of *LNCS*, Lisboa, Portugal, Aug. 2005. Springer Verlag.

[7] G.-P. C. and C. J. Web Services and Web Service Security Standards. *Information Security Technical Report*, 10(1):15–24, 2005. Elsevier.

[8] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computations*. Research Monographs in Parallel and Distributed Computing. Pitman, 1989.

[9] M. Cole. Bringing Skeletons out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming. *Parallel Computing*, 30(3):389–406, 2004.

[10] Coregrid home page, 2006. http://www.coregrid.net.

[11] M. Danelutto. QoS in parallel programming through application managers. In *Proceedings of the 13th Euromicro Conference on Parallel, Distributed and Network-based processing*. IEEE, 2005. Lugano.

[12] M. Danelutto and M. Dazzi. Joint Structured/Unstructured Parallelism Exploitation in Muskel. In *Proceedings of ICCS 2006 / PAPP 2006*, LNCS. Springer Verlag, May 2006. to appear.

[13] J. Dünnweber and S. Gorlatch. HOC-SA: A grid service architecture for higher-order components. In *IEEE International Conference on Services Computing, Shanghai, China*, pages 288–294. IEEE Computer Society Press, September 2004.

[14] A. A. et al. Survey Material on Trust and Security, 2005. Deliverable D.IA.03, CoreGRID, www.coregrid.org.

[15] Globus web site. http://www.globus.org.

[16] H. Kuchen. A Skeleton Library. In *Euro-Par 2002, Parallel Processing*, number 2400 in LNCS, pages 620–629. Springer Verlag, August 2002.

[17] ProActive home, 2006. www-sop.inria.fr/oasis/ProActive/.

[18] J. Sérot and D. Ginhac. Skeletons for parallel image processing : an overview of the SKiPPER project. *Parallel Computing*, 28(12):1785–1808, Dec 2002.

[19] D. Snelling and K.-J. et al. Next Generation Grids 2 – Requirements and Options for European Grids Research 2005-2010 and Beyond, 2004.

[20] M. Vanneschi. The Programming Model of ASSIST, an Environment for Parallel and Distributed Portable Applications. *Parallel Computing*, 12, December 2002.

[21] V. Welch, F. Siebenlist, I. Foster, J. Bresnahan, and K. C. et al. Security for Grid Services. In *Proceedings of 12th IEEE International Symposium on High Performace Distributed Computing*. IEEE Computer Society Press, 2003.

15th EUROMICRO International Conference on Parallel, Distributed and Network-Based Processing (PDP'07)
0-7695-2784-1/07 $20.00 © 2007 IEEE

IEEE COMPUTER SOCIETY