Design and Implementation of a Service-integrated Session Layer for Efficient Message Passing in Grid Computing Environments

Carsten Clauss, Stefan Lankes, Thomas Bemmerl
Chair for Operating Systems, RWTH Aachen University
Kopernikusstr. 16, 52056 Aachen, Germany
{clauss, lankes, bemmerl}@lfbs.rwth-aachen.de

Abstract

When running large parallel applications with demands for resources that exceed the capacity the local computing site offers, the deployment in a distributed Grid environment may help to satisfy these demands. However, since such an environment is a heterogeneous system by nature, there are some drawbacks that, if not taken into account, are limiting its applicability. First of all, one has to apply a meta-computing or Grid-enabled message-passing library in order to have the ability to route messages to remote sites as well as still being able to exploit fast site-local network facilities. Then, because the inter-site communication usually constitutes the system's bottleneck, appropriate quality of service parameters should be provided and policed for those connections during the application's execution. And finally, the parallel runtime environment of the distributed application should offer service interfaces in order to enable its interaction with Grid middleware. In this paper, we present a new library called ISI whose functionalities meet those requirements in terms of a session layer to be integrated into Grid-enabled message-passing implementations.

1. Introduction and Motivation

The solution of critical numerical problems often exceeds the computing and memory capacity a local computing site offers. Therefore, by combining distributed computing resources, as for example provided by computational Grid environments [9], can help to satisfy the resource demands such large applications desire. Advances in wide-area networking technology have fostered this trend towards geographically distributed high-performance parallel computing in the recent years. However, as Grid resources are usually heterogeneous by nature, this is also true for the communication characteristics. Especially the inter-site communication often constitutes a bottleneck in terms of higher latencies and lower bandwidths than compared to the site-internal case. The reason for this is that the inter-site communication is typically handled via wide-area transport protocols and respective networks, whereas the internal communication is conducted via fast local-area networks or even via dedicated high-performance interconnections. That in turn means that an efficient utilization of such a hierarchical and heterogeneous infrastructure demands a communication middleware that provides support for all these different kinds of networks and transport protocols.

Grid-enabled MPI Since MPI [18] is the most important API for implementing parallel programs for large-scale environments, also some MPI libraries have already been extended in order to meet these demands of distributed and heterogeneous computing. Those libraries are often called *Grid-enabled* because they do not only use plain TCP/IP (which is obviously the lowest common denominator) for all inter-process communication, but are also capable of exploiting fast but local networks and interconnect facilities in order to accommodate the Grid's hierarchy. Hence, for being able to provide support for the various high-performance cluster networks and their specific communication protocols, most of those libraries in turn rely on other high-level communication libraries (like site-*native* MPI libraries), rather than implementing this support inherently. Therefore, such a Grid-enabled MPI library can be understood as a kind of a *meta-layer* that bridges the distributed computing sites, and for that reason their application area is also referred to as a so-called *meta-computing* environment. The most common meta-computing and Grid-enabled MPI libraries are MPICH-G2 [15], PACX-MPI [1], GridMPI [16], StaMPI [25] and MetaMPICH [21], which are all proven to run large-scale applications in distributed environments. Although these meta-MPI implementations usually use native MPI support for site-internal communication, as for example provided by a site-local vendor MPI, they must also be based on at least a transport layer being capable of wide-area communication for bridging and forwarding messages also to the remote sites. However, since regular transport protocols like TCP/IP are commonly point-to-point-oriented, it is a key task of such a bridging layer to setup all the required inter-site connections and thus to act like a *session layer* for the wide-area communication.

Grid-specific Transport Protocols Obviously, TCP/IP is the standard transport protocol used in the Internet, and due to its general design, it is also often employed in Grid environments. However, it has been proven that TCP has some performance drawbacks especially when being used in high-speed wide area networks with high-bandwidth but high-latency characteristics [8]. Hence, Grid environments, which are commonly based on such dedicated high-performance wide area networks, often require customized transport protocols that take the Grid-specific properties into account. [31, 28]

Since a significant loss of performance arises from TCP's window-based congestion control mechanism, several alternative communication protocols like FOBS [6], SABUL [10], UDT [11] or PSockets [23] try to circumvent this drawback by applying their own transport policies and tactics at application level. That means that they are implemented in form of user-space libraries which in turn have to rely on standard kernel-level protocols like TCP or UDP, again. An advantage of this approach is that there is no need to modify the network stack of the operating systems being used within the Grid. The disadvantage is, of course, the overhead of an additional transport layer on top of an already existing network stack. Nevertheless, a further advantage of such user-space communication libraries is the fact that they can offer a much more comprehensive and customized interface to the Grid applications than the general purpose OS socket API does.

However, in the recent years, a third kernel-level transport protocol has become common and available (at least within the Linux kernel): the Stream Control Transmission Protocol (SCTP) [24, 30] which provides, similar to TCP, a reliable and in-sequence transport service. Additionally, SCTP offers several features not present in TCP, as for example the *multihoming* support. This means that an endpoint of a SCTP association (SCTP uses the term “association” to refer to a connection) can be bound to more than one IP address at the same time. Thus, a transparent failover between redundant network paths becomes possible. Furthermore, it can be shown that SCTP *may* also perform much better than TCP especially in heterogeneous wide-area networks due to a faster congestion control recovery mechanism [22, 19]. For that reasons, employing SCTP also in Grid environments may be beneficial compared to common TCP. [14, 4]

When looking at this diversity of alternative transport protocols, the question arise which one should be used by the bridging session layer of a message-passing library in meta-computing environments? The answer is that this depends on the properties of the actual environment. In fact, due to the Grid's heterogeneous nature, the best solution may differ even within the Grid. Moreover, since Grid resources can be volatile, e.g. an initially assigned bandwidth does not necessarily be granted during a whole session, the optimal protocol to be used may also vary in the course of time. For that reason, an efficient session layer for message-passing-based Grid computing should be capable of supporting more than one transport facility at the same time. Nevertheless, such a session layer should also be aware of the inter-site communication overhead by being and acting as resource-friendly as possible in this respect.

Demand for Integrated Services In order to exploit a Grid environment at its full potential, the underlying network must be a managed resource, just like computing and storage resources usually are. As such, it should be manages by an intelligent and autonomic Grid middleware. [28, 13] Such a middleware, like a Grid scheduler, needs to retrieve runtime information about the current capacity and quality of the communication infrastructure, as well as information about the communication patterns and characteristics of the running Grid applications. For that purpose, the possibility of a dynamic interaction between such a scheduling middleware and the respective application would be very desirable. Therefore, a favored session layer would also provide *Grid service interfaces* in order to make such information inquirable at runtime. Moreover, a dedicated interface that also allows to access and even to reconfigure the session settings at runtime would help to exploit the Grid's heterogeneous network capabilities at their best. Consequently, a session layer for an actual efficient meta-computing-related message-passing should provide such integrated services to the Grid environment.

Remainder of the Paper The remainder of the paper is organized as follows: In Section 2, we detail the design and the implementation of such a session layer that meets all these issues and requirements described above. We will show how the actual session establishment is performed and how payload transfers are handled via the established connections. The design of an integrated service interface for providing dynamic interaction with other Grid middleware is presented in Section 3. Finally, the description of an already successfully conducted application example and an outlook on possible future enhancements and planned follow-up projects conclude the paper in Section 5.

2. Implementation of the Session Layer

In this section, we want to detail the implementation of the service-integrated session layer called *ISI*[1]. In its capacity as a session layer, ISI is located according to the OSI model [5] between the upper presentation layer and the lower transport layer, as it is illustrated in Figure 1. However, its actual implementation is realized in the form of a library that has recently been developed at our institute.

Since ISI has been especially designed as a session layer for message passing in meta-computing and Grid environments, the upper presentation layer is principally represented by the higher-level MPI functionalities. Whereas, on the other hand, the interfaces to the lower transport layer are traditionally abstracted by the concept of *network sockets*. Thus, one key task of this session layer is to establish the required connections between the distributed processes of a heterogeneous MPI environment by mapping this connection establishment onto socket-related client-server-semantics.

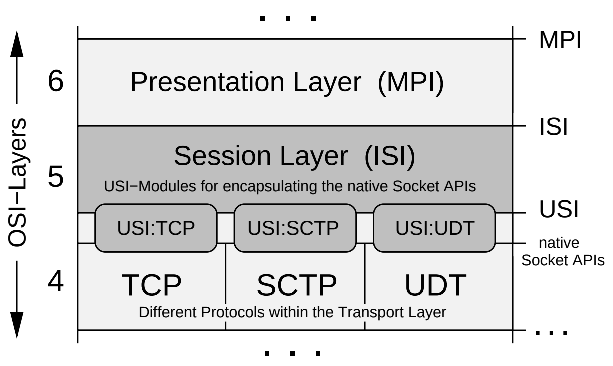

Figure 1. Classification According to OSI

However, in order to support a more generalized and flexible interface design for the ISI layer, we have decided not to base the layer directly upon the native socket interfaces provided by the respective transport protocols, but rather to introduce a further abstraction layer called *USI*.[2] The purpose of this small layer is to unify the access to the lower layers within the ISI implementation, while still being able to utilize the different features each of the transport protocols may offer. This goal is achieved by encapsulating the calls to the respective socket interfaces in so-called *USI-Modules* and hence providing one module instance per each supported transport protocol, as it is also denoted in Figure 1.

[1]Although *ISI* is rather a name than an actual abbreviation, one may interpret the letters as a multi-layered acronym for *Service-Integrated Session layer Implementation on a Socket Interface*.

[2]Here, *USI* is an abbreviation for *Uniform Socket Interface*

2.1. The Domain Configuration

Within the ISI terminology, an established (logical) communication session between the participating processes is represented by a so-called *ISI-Domain*. Such a domain is characterized by the affiliation and the number of covered processes. Whereas, in turn, each contained process is characterized by a unique rank within the domain, just in analogy to an MPI world. However, the correlation between a respective process and its rank is identified via a special *domain configuration* in case of an ISI session.

In this context, such a domain configuration is an XML-coded [27] dataset that has to be passed to each participating process in a manner to be determined during the domain initialization. Therefore, a certain access method needs to be specified in order to enable the actual access for the processes to the needed configuration dataset. Here, possible access methods may be the readout of a local configuration file or a query on a central configuration server.

After its retrieval, the configuration dataset gets parsed so that the required information like *domain size*, the respective *domain rank* and the necessary *address information* (see next Section) become available at all participating processes.

At this point it should be emphasized that all actual connections between the processes within a session are just established on demand. The reason for this is that in a large Grid environment with *many* processes an establishment of all imaginable N-to-N connections during the domain initialization would constitute an inappropriate overhead. However, a once-established connection will usually be retained until the termination of the processes.

```
<domain identifier="my_domain">
    <size> 2 </size>
    <process identifier="host-01">
        <rank> 0 </rank>
    </process>
    <process identifier="host-02">
        <rank> 1 </rank>
    </process>
    . . .
</domain>
```

Figure 2. Excerpt of a Domain Configuration

Figure 2 shows a simple example of an ISI configuration dataset describing a session with just two participating processes. As one can see, during parsing this configuration, the processes can recognize their domain affiliation and their respective domain rank by means of so-called *domain and process identifiers*. For that purpose, matching identifiers need to be passed to the domain initialization function in terms of string parameter. Here, for instance, the *username*, the *name of the application* and/or the *hostnames* can serve as such identifiers being detectable by each process on its own.

2.2. Handling of Payload Transfers

Although it is not a key task of a session layer to process the actual transfer of payload, it still needs to provide the possibility to associate a communication request with an established connection. For that purpose, the ISI layer offers *send and receive functions* that serve as such entry points for communication requests posted by the upper layer.

Those functions are designed in a quite straightforward-manner, what means that on the one hand, they are referring to the semantics of the common non-blocking MPI send and receive functions. On the other hand, those functions should just act as a gateway that maps domain and rank related communication requests onto the underlying established socket connections between the processes. In this context it should be mentioned that the ISI layer itself works strictly *header-less*. That means that the ISI layer does not add any additional management data to the payload, but just gathers and passes the posted references of message buffers to the lower transport layer.

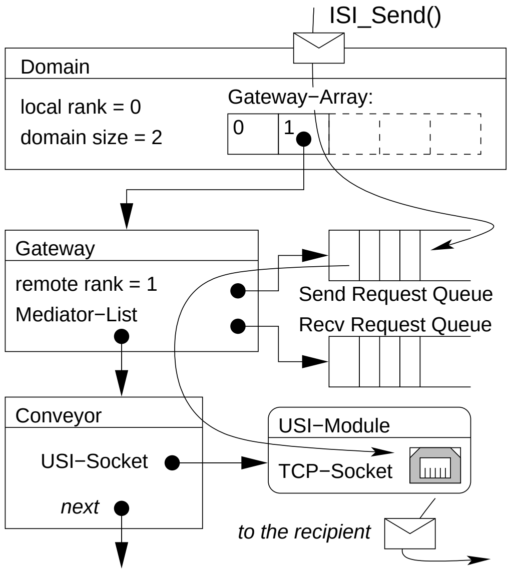

Figure 3. Processing of Payload Requests

For that purpose, each domain instance holds an array of so-called *Gateway* objects (with one object per remote domain rank) that features appropriate *FIFO queues* for handling the posted requests, as it is denoted in Figure 3. In this example, the process with rank 0 posts a send request for a message to be received by process 1. As one can see, the request is directed across the Gateway-Array and the respective send queue to an established transport layer connection, here constituted by a TCP socket.

As already mentioned, the actual interfaces to the respective transport protocols are encapsulated in the so-called USI-Modules. Since those modules are in turn embedded into instanceable objects, different transport protocols can be used within one ISI session. In fact, it is possible that for example some processes want communicate via TCP while others may find SCTP as the best choice, and all within one ISI-Domain. For that purpose, the domain configuration must contain corresponding *module and address information* in order to establish the required connections based on the desired transport protocols.

```
<addresses rank="1">
    <connector module="TCP">
        123.123.123.123:12345
    </connector>
    <connector module="SCTP">
        123.123.123.123:45678|
        231.231.231.231:56789
    </connector>
</addresses>
```

Figure 4. Excerpt of an Address Information

Figure 4 shows an exemplary excerpt of such a multi-protocol domain configuration. In this snippet, the information is coded how to establish a connection with the process holding rank 1. As one can see, this rank provides two USI-Modules and thus will listen on the two stated network addresses. While the first one of these address entries refers to a TCP connection, the second entry here represents a multihomed endpoint of a SCTP association.

Currently, the ISI implementation provides USI-Modules for TCP, SCTP and UDT. Others, like a *PSockets* [23] based module or a module being customized especially for the extended *WinSock*-API [17] may follow.

However, when looking again at Figure 3, one may consider a yet not explained object called *Conveyor*. This object is derived from an abstract superclass called *Mediator* and represents in ISI terminology an actual established payload connection. Each ISI-Gateway posses a *list* containing such Mediator-derived objects like Conveyors, and consequently there can exist more than one payload connection between two processes at the same time.

At this point the question arises, which of these possibly multiple connections should be used for the actual payload transfer. Until now, a (logical) connection is characterized by a certain pair of processes, a chosen transport protocol represented by an USI-Module and the respective address information. However, for the purpose of *judging* a connection's *quality*, also additional attributes like latency or bandwidth can be associated with the Conveyor objects within an ISI-Domain. In fact, it is the task of additional integrated services within the ISI layer to obtain such information and to ensure their consideration during session establishment and management. Those services and their interfaces will be explained in the next section.

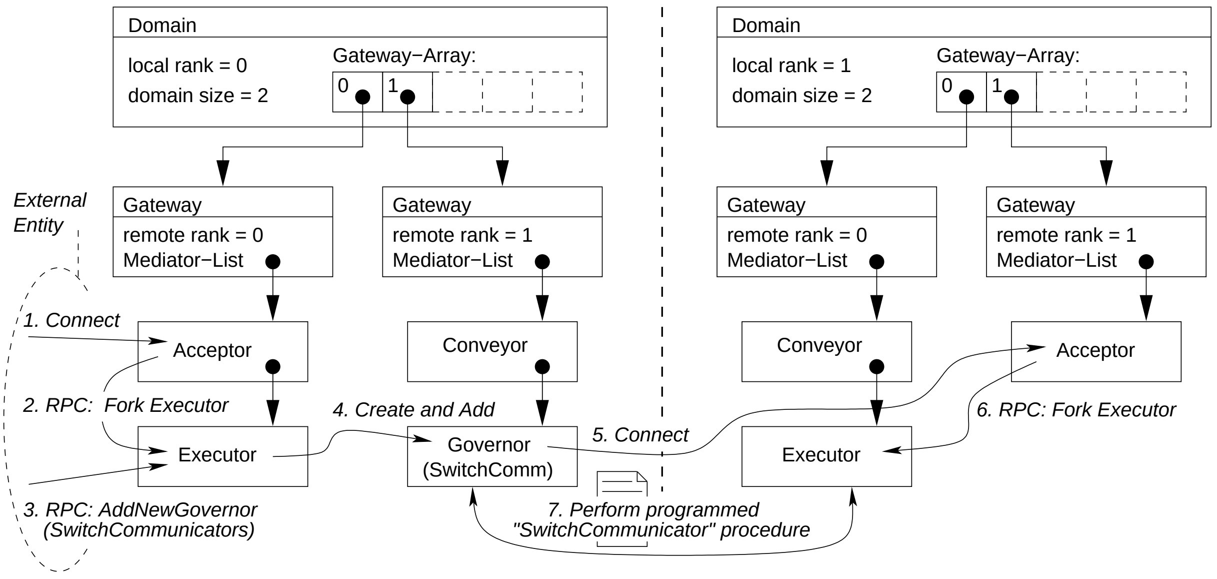

Figure 5. An Example for Deploying Governor Objects Committed by an External Entity

3. The Integrated Services

All services offered by the ISI layer can be accessed by the Grid environment via *remote procedure calls* (RPC). Although there exist several approaches for implementing RPC facilities in Grid environments [20, 26, 12], we have decided to base our implementation on the raw XML-RPC [29] specification. Therefore, all RPC queries have to be handled via XML-coded remote method invocations of so-called *ISI-Service-Objects*.

In turn, such service objects are instances of *Mediator*-derived classes, such as the above mentioned Conveyor objects. And since the offered services are typically rank-related, the service objects are stored in the Mediator-List of the respective ISI-Gateways, just like the Conveyor objects, too. All these objects can be understood as finite state automata that are triggered by outgoing or incoming messages. Thus, in case of a Conveyor, such messages are just the payload requests, while in case of service objects, those messages are usually constituted by the XML-coded remote method calls and responses. Consequently, service inquiries on an established ISI-Domain are also handled via socket connections provided by the USI-Modules. That means that also the RPC-related communication can utilize the different supported transport protocols. Although the state transitions of each service object are triggered by events like message arrivals, the respective event handling is scheduled within the ISI layer in an *intelligent* way. Thus, for example, payload requests are handled with a higher priority than service inquiries. On the other hand, an intelligent scheduling policy also protects service inquiries from starvation.

3.1. Reporting Services

Simple services just provide the caller with status information about the current session, as for instance whether a certain connection has already been established, which protocol is (or should be) in use, or how many bytes of payload have already been transfered on this connection. These information can then be evaluated by an external entity like a Grid monitoring daemon in order to detect bottlenecks or deadlocks in communication. For the purpose of being accessible for such external entities, each process within an ISI session provides so-called *Acceptor* objects which are listening on announced addresses for such incoming inquiries.

However, those addresses may be just the same as for accepting connections from other processes within an ISI-Domain. For that reason, each connecting process must initially tell the Acceptor whether it is a process that exhibits a domain rank, or if it is about an external monitoring instance. Thus, since the remote rank of a connecting process (that is an incoming connection) is not predictable *a-priori*, Acceptors are stored within the Mediator-List of that Gateway object being indexed by the *local* rank.

3.2. Intervening Services

Besides such query-related services, each ISI process also offers RPC interfaces that allow external entities actually to *govern* the behavior and the session-related state of that process. That way, a monitoring or scheduling instance is given the ability to reconfigure an already established session even at runtime. For this purpose, a respective master entity has to send RPC-related commands to the ISI processes which will then be carried out by so-called ISI-*Executor* objects within the processes. Upon receiving of such a command, an Executor may change, for example, the active Conveyor of a Gateway object and thus the transport protocol being used for payload transfers.

However, as one can imagine, the change of the actual payload connection is rather a process-*pair*-related command, than a command to be executed on a single process. Due to this reason, a further Mediator-derived object is the so-called *Governor*. Actually, such a Governor is a kind of a *program* which is based on ISI-Executor-related commands and responses. That means that a Governor can be deployed to let one ISI process control the behavior of another. Since there a various useful courses of action within such a programmed remote control thinkable, "*Governor*" is actually the name of a superclass of derived objects implemented for performing different tasks.

For example, such a task may be the synchronized switching of a payload connection to another transport protocol. Processing this task, the Governor must initially ensure that all further enqueueing of payload requests gets temporarily stopped on both sites. Then the master has to determine whether the number of bytes of the local and the remote pending requests are identical (or zero). This is because only in this case a proper mapping of payload data being still in transit onto the posted receive requests can be assured afterwards. In case this condition is not met, the Governor can try to obtain such a match by regulating the request enqueueing rate on both sites. Eventually, the Governor can safely enable the new payload connection, while possibly pending older requests may still be satisfied by incoming data from the former connection.

If an external entity wants to invoke such a connection switching procedure between two ISI processes, it just needs to connect to one of them and to send the rank-related Executor command "*AddNewGovernor*" with "*SwitchConveyors*" as argument, whereupon the process will create such a new Governor object to be stored in the Mediator-List of the respective Gateway object. This Governor will then connect to an Acceptor at the remote rank that forks an Executor which in turn will carry out the required measures, guided by the Governor, to perform a proper switching of the connections. Figure 5 shows an example for such a procedure where an external entity commits a Governor in order to perform such a pair-related task like *SwitchConveyors*.

3.3. Monitoring Services

However, besides such connection-related control mechanisms based on Governors, also self-referring monitoring services are supported by the ISI layer to the participating domain processes themself. These services are represented in ISI terms by means of so-called *Supervisor* objects. Such objects, though also derived from the Mediator superclass, are not associated with a connection to a remote rank. For that reason, they are usually stored in the Mediator-List of the Gateway object being located at the index of the own rank inside the domain's Gateway-Array, just like Acceptors.

Although they do not refer to a distinct remote rank, the Supervisor services can, in turn, initiate connection-related actions by means of creating and deploying appropriate Governor objects. Such actions may be triggered, for example, by the detection of a timeout, a bottleneck or the requirement of a cleanup. At this point it should be mentioned that also external entities have the ability to control, add or just delete such self-monitoring Supervisors for a certain rank at runtime. By this means, a Supervisor can act as a sort of a remote daemon deployed by an external entity that monitors the respective process and that can trigger the corresponding actions *on-site*, this means without any further external interaction or intervention. For that purpose, the external entity has just to act in almost the same manner as already described for the case of deploying Governor objects, though with the difference that "*AddNewSupervisor*" is the RPC command to be used.

An example of an actual Supervisor object is the *LazyAutoConnect* service that realizes an *on-demand* connection establishment. For this purpose, such an object frequently checks the queues of unconnected Gateways for posted payload requests and then triggers the respective connect/accept procedure if necessary.

3.4. Benchmarking Services

Finally, the question remains how quality-of-service metrics like latency and bandwidth of a connection can be obtained within an ISI session. For that purpose, the ISI layer offers special *Surveyor*-Service-Objects which are quite similar to Conveyor objects, with the difference that they are not intended for an actual payload transfer but for measuring the connection characteristics by means of data transfer being performed especially for benchmarking purpose. Although the data being sent and received within such a benchmarking procedure *can* be constituted by actual payload being posted in terms of requests from a higher layer, the Surveyor objects can also generate *faked data* for that purpose. That in turn means that such a benchmarking procedure can interfere with regular payload traffic within an ISI session and hence should be kept as short as possible. Nevertheless, the gained information about a connection's quality (and resulting activities like protocol switching) can still contribute to a better overall performance of an ISI session.

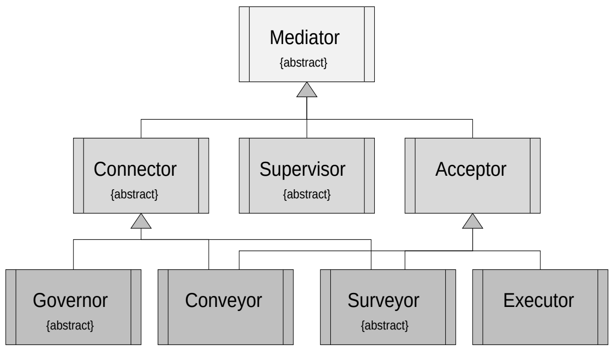

Figure 6. Overview of the ISI-Service-Objects

3.5. Further Services

Concluding, Figure 6 shows the derivation tree displaying the hierarchy of the described ISI-Service-Objects. As one can see, the connection-related objects like Governors, Executors, Conveyors and Surveyors inherit from a Connector and/or the Acceptor class, whereas a Supervisor is directly derived from the Mediator superclass. Here it should be emphasized that Governor, Supervisor and Surveyor are still *abstract* classes, while further subclasses (not displayed in Figure 6) implement the actual behavior of the respective services, as for example "*SwitchConveyors*" is a particular service instance of a Governor.

Those further subclasses are all implemented in terms of dynamically loaded objects located in shared libraries. That means that when creating such a service instance, for example by emitting an "*AddNewGovernor*" RPC command, the name of the requested Governor instance, given as a RPC parameter, serves as a handle for loading the respective object. That way, the handling of such objects is kept as general as possible within the ISI layer for the benefit of flexibility.

4. Integration into MetaMPICH

As already pointed out in the motivation section, the ISI session layer has not been developed with the intention of being a stand-alone communication library, but has been especially designed to be integrated into meta-computing and Grid-enabled MPI libraries. Therefore, our overall desired goal would be a comprehensive integration of ISI into several meta-computing-related projects. However, as a first feasibility study, we have already successfully integrated ISI's functionalities into a project called *MetaMPICH*, which is about the implementation and enhancement of a Grid-enabled MPI library and which has also been conducted at our institute. [21]

In order to meet the demands of heterogeneous Grid environments, this MPI library provides even two different methods for linking the remote sites: a *router-based* method and a so-called *multi-device* method [3]. In particular the multi-device method of MetaMPICH has significantly benefited from an ISI managed session establishment. This is because the former version of this method was based on pure TCP connections that had all been established statically during initialization. By now, MetaMPICH can also provide the different transport protocols supported by the USI-Modules of the ISI library, whereas the (actual) connection establishment is done *on-demand* in a dynamic and resource-saving manner.

Besides this feature of supporting multiple transport protocols in a very flexible manner, further key features of ISI are its integrated services. Although MetaMPICH does not enforce the usage of a specific Grid middleware (in fact, it can be used without such a middleware at all), its runtime system has already been extended by the ability to interact with a so-called *meta-scheduling* service in UNICORE-based Grid environments [7, 2]. We believe that in future the interaction between MetaMPICH (or its successor) and other Grid middleware such as UNICORE can particularly benefit from ISI's service-related interfaces.

5. Conclusions and Outlook

In this paper, we have presented the design of the ISI session layer library. We have pointed out the demand for its capability of supporting Grid-specific transport protocols, as well as the demand for its integrated service interfaces in order to provide a dynamic interaction with the outer Grid-environment. After detailing ISI's implementation in terms of the ISI-Domain abstraction concept and the multiplicity of ISI-Service-Objects, we have presented a successfully conducted application example.

In order to keep this paper focused, we are not able to cover all aspects of interest within the broad scope of an efficient, reliable and secure session layer. In particular, security aspects like authentication and authorization are major topics, but not discussed here. Furthermore, we have to conclude the paper, due to lack of space, without presenting any performance results. In fact, we have spent a lot of efforts in optimizing the performance by shortening the critical paths. In addition, a comparison of the presented architecture with other approaches for Grid-enabled MPI solutions would be desirable, too. Therefore, we aim to cover all these aspects in future work and publications.

Finally, we want to emphasize that ISI is a free library and thus may in future also be utilized by other meta-computing projects than those being conducted at our institute. Actually, we think that an integration even into external projects should also prove to be quite easy, because of ISI's modular and service-oriented design.

References

[1] T. Beisel, E. Gabriel, M. Resch, and R. Keller. Distributed Computing in a Heterogeneous Computing Environment. In *Proceedings of the 5th European PVM/MPI Users' Group Meeting (EuroPVM/MPI)*, UK, September 1998. Springer.

[2] B. Bierbaum, C. Clauss, T. Eickermann, L. Kirtchakova, A. Krechel, S. Springstubbe, O. Wäldrich, and W. Ziegler. Orchestration of distributed MPI-Applications in a UNICORE-based Grid with MetaMPICH and MetaScheduling. In *Proceedings of the European PVM/MPI Users' Group Meeting (EuroPVM/MPI)*, Germany, September 2006. Springer.

[3] B. Bierbaum, C. Clauss, M. Pöppe, S. Lankes, and T. Bemmerl. The new Multidevice Architecture of MetaMPICH in the Context of other Approaches to Grid-enabled MPI. In *Proceedings of the European PVM/MPI Users' Group Meeting (EuroPVM/MPI)*, Germany, September 2006. Springer.

[4] O. Buyanjargal, J. T. Kim, S. Y. Lee, and Y. Kwon. Performance Improvement of Grid Web Services based on Multi Homing Transport Layer. In *Computer and Information Technology (CIT)*. IEEE Computer Society, 2006.

[5] D. Day and H. Zimmerman. The OSI Reference Model. *Proceedings of the IEEE*, Vol. 71(12):1334–1340, 1983.

[6] P. M. Dickens. FOBS: A Lightweight Communication Protocol for Grid Computing. In *Processing of the 9th International Euro-Par Conference (Euro-Par'03)*, Austria, August 2003.

[7] D. W. Erwin and D. F. Snelling. UNICORE: A Grid Computing Environment. *Lecture Notes in Computer Science*, 2150:825ff, 2001.

[8] W. Feng and P. Tinnakornsrisuphap. The Failure of TCP in High-Performance Computational Grids. In *Proceedings of Supercomputing'2000 (CD-ROM)*, Dallas, November 2000. IEEE and ACM SIGARCH.

[9] I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a Future Computing Infrastructure*. Morgan Kaufmann, 1999.

[10] Y. Gu and R. L. Grossman. SABUL: A Transport Protocol for Grid Computing. *Journal of Grid Computing*, 1(4):377–386, 2003.

[11] Y. Gu and R. L. Grossman. UDT: UDP-based data transfer for high-speed wide area networks. *Computer Networks*, 51(7):1777–1799, 2007.

[12] M. Gudgin, M. Hadley, N. Mendelsohn, J.-J. Moreau, and H. F. Nielsen. SOAP version 1.2. W3C recommendation, W3C, June 2003.

[13] S. Hessler and M. Welzl. Seamless Transport Service Selection by Deploying a Middleware. *Computer Communications*, 30(3):630–637, 2007.

[14] H. Kamal, B. Penoff, and A. Wagner. SCTP-based Middleware for MPI in Wide-Area Networks. In *Communication Networks and Services Research Conference (CNSR)*. IEEE Computer Society, 2005.

[15] N. Karonis, B. Toonen, and I. Foster. MPICH-G2: A Grid-enabled Implementation of the Message Passing Interface. *Journal of Parallel and Distributed Computing*, 63(5):551 – 563, 2003.

[16] M. Matsuda, Y. Ishikawa, Y. Kaneo, and M. Edamoto. Overview of the GridMPI Version 1.0. In *Proceedings of the SWoPP05, Japan*, 2005.

[17] Microsoft. *Windows Sockets 2 Application Programming Interface, An Interface for Transparent Network Programming Under Microsoft Windows*. Manual.

[18] MPI Forum. MPI: A Message-Passing Interface Standard. *International Journal of Supercomputing Applications*, 1994.

[19] D. Nagamalai, S.-H. Lee, W. G. Lee, and J.-K. Lee. SCTP over High Speed Wide Area Networks. In *Proceedings of the 4th International Conference on Networking (ICN)*, France, April 2005. Springer.

[20] H. Nakada, S. Matsuoka, K. Seymour, J. Dongarra, C. Lee, and H. Casanova. GridRPC: A Remote Procedure Call API for Grid Computing. In *Proceedings of the 3rd International Workshop on Grid Computing (GRID)*, USA, November 2002. Springer.

[21] M. Pöppe, S. Schuch, and T. Bemmerl. A Message Passing Interface Library for Inhomogeneous Coupled Clusters. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, France, April 2003.

[22] R. Rajamani, S. Kumar, and N. Gupta. SCTP versus TCP: Comparing the Performance of Transport Protocols for Web Traffic. Free Whitepaper from University of Wisconsin, July 2002.

[23] H. Sivakumar, S. Bailey, and R. L. Grossman. PSockets: The Case for Application-level Network Striping for Data Intensive Applications using High Speed Wide Area Networks. In *Proceedings of Supercomputing 2000*, Dallas, November 2000. IEEE and ACM SIGARCH.

[24] R. Stewart and Q. Xie. Stream Control Transmission Protocl (SCTP) - A Reference Guide, 2001.

[25] H. K. Toshiyuki Imamura, Yuichi Tsujita and H. Takemiya. An Architecture of StaMPI: MPI Library on a Cluster of Parallel Computers. In *Proceedings of the European PVM/MPI Users' Group Meeting (EuroPVM/MPI)*, Hungary, 2000. Springer.

[26] S. Verma, M. Parashar, J. Gawor, and G. von Laszewski. Design and Implementation of a CORBA Commodity Grid Kit. In *Proceedings of the Second International Workshop on Grid Computing (GRID)*, USA, November 2001. Springer.

[27] W3C. *Extensible Markup Language (XML) 1.0 (Fourth Edition)*, September 2006.

[28] M. Welzl and M. M. Yousaf. Grid-Specific Network Enhancements: A Research Gap? In *International Workshop on Autonomic Grid Networking and Management (AGNM'05)*, Spain, October 2005. IEEE.

[29] D. Winer. *XML-RPC Specification*. UserLand, Inc., June 1999.

[30] L. M. Yarroll and K. Knutson. Linux Kernel SCTP: The Third Transport. Whitepaper.

[31] Y. Zhu, A. Bassi, P. Massonet, and D. Talia. Mechanisms for High Volume Data Transfer in Grids. Technical report, Institute on Knowledge and Data Management, CoreGRID – Network of Excellence, December 2007.