Editorial Board

Ozgur Akan
Middle East Technical University, Ankara, Turkey
Paolo Bellavista
University of Bologna, Italy
Jiannong Cao
Hong Kong Polytechnic University, Hong Kong
Falko Dressler
University of Erlangen, Germany
Domenico Ferrari
Università Cattolica Piacenza, Italy
Mario Gerla
UCLA, USA
Hisashi Kobayashi
Princeton University, USA
Sergio Palazzo
University of Catania, Italy
Sartaj Sahni
University of Florida, USA
Xuemin (Sherman) Shen
University of Waterloo, Canada
Mircea Stan
University of Virginia, USA
Jia Xiaohua
City University of Hong Kong, Hong Kong
Albert Zomaya
University of Sydney, Australia
Geoffrey Coulson
Lancaster University, UK
Report on GridNets 2008

Beijing, China October 8–10, 2008

The Second International Conference on Networks for Grid Applications
In cooperation with ACM SIGARCH
Sponsored by ICST
Technically co-sponsored by Create-Net and EU-IST

The GridNets conference series is an annual international meeting which provides a focused and highly interactive forum where researchers and technologists have the opportunity to present and discuss leading research, developments, and future directions in the grid networking area. The objective of this event is to serve as both the premier conference presenting the best grid networking research and a forum where new concepts can be introduced and explored.

After the great success of last year's GridNets in Lyon, France, which was the first "conference" event, we decided to move GridNets to Beijing, China in 2008. We received 37 papers, and accepted 19. For this single-track conference, there were 2 invited keynote speakers, 19 reviewed paper presentations, and 4 invited presentations. This program was supplemented by a workshop on Service-Aware Optical Grid Networks and a workshop on Wireless Grids; both of these workshops took place on the first day of the conference day.

Next year's event is already being planned, and it will take place in Athens, Greece. We hope to see you there!

Tomohiro Kudoh
Pascale Vicat-Blanc Primet
Joe Mambretti
Organization

Steering Committee

Imrich Chlamtac, Chair, CREATE-NET
Pascale Vicat-Blanc Primet, INRIA, ENS-Lyon (France)
Michael Welzl, University of Innsbruck (Austria)

General Co-chairs

Chris Edwards, Lancaster University (UK)
Michael Welzl, University of Innsbruck (Austria)
Junsheng Yu, Beijing University of Posts and Telecommunications (China)

Local Co-chairs

Yuan'an Liu, Beijing University of Posts and Telecommunications (China)
Tongxu Zhang, China Mobile Group Design Institute Co. Ltd. (China)
Shaohua Liu, Beijing University of Posts and Telecommunications (China)

Program Committee Co-chairs

Pascale Vicat-Blanc Primet, INRIA, ENS-Lyon (France)
Joe Mambretti, Northwestern University (USA)
Tomohiro Kudoh, AIST (Japan)

Publicity Co-chairs

Serafim Kotrotsos, for Europe, EXIS IT (Greece)
Xingyao Wu, for Asia, China Mobile Group Design Institute Co. Ltd. (China)
Sumit Naiksatam, for USA, Cisco Systems

Publications Chair

Marcelo Pasin, INRIA, ENS-Lyon (France)
email: marcelo.pasin@ens-lyon.fr
Exhibits and Sponsorship Co-chairs
Peng Gao China Mobile Group Design Institute Co. Ltd. (China)
Junsheng Yu Beijing University of Posts and Telecommunications (China)

Workshops Chair
Ioannis Tomkos Athens Information Technology (Greece)

Industry Track Chair
Tibor Kovács ICST

Panels Chair
Chris Edwards Lancaster University (UK)

Conference Organization Chair
Gergely Nagy ICST

Finance Chair
Karen Decker ICST

Webmaster
Yehia Elkhatib Lancaster University (UK)

Technical Program Co-chairs
Pascale Vicat-Blanc Prime INRIA, ENS-Lyon (France)
Joe Mambretti Northwestern University (USA)
Tomohiro Kudoh AIST (Japan)

Technical Program Members
Micah Beck University of Tennessee (USA)
Augusto Casaca INESC-ID (Portugal)
Piero Castoldi SSSUP (Italy)
Cees De Laat University of Amsterdam (Holland)
Chris Develder, Ghent University - IBBT (Belgium)
Silvia Figueira, Santa Clara University (USA)
Gabriele Garzoglio, Fermi National Accelerator Laboratory (USA)
Olivier Glück, INRIA, ENS-Lyon (France)
Paola Grosso, University of Amsterdam (Holland)
Yunhong Gu, University of Illinois (USA)
Wei Guo, Shanghai Jiao Tong University (China)
David Hausheer, University of Zurich (Switzerland)
Guili He, China Telecommunication Technology Lab (China)
Doan Hoang, University of Technology, Sydney (Australia)
Tao Huang, Institute of Software, Chinese Academy of Sciences (China)
Yusheng Ji, NII (Japan)
Raj Kettimuthu, Argonne National Laboratory (USA)
Oh-kyoung Kwon, KISTI (Korea)
Dieter Kranzlmueller, GUP-Linz (Austria)
Francis Lee Bu Sung, Nanyang Technological University (Singapore)
Shaohua Liu, Beijing University of Posts and Telecommunications (China)
Tao Liu, China Mobile Group Design Institute Co. Ltd. (China)
Tiejun Ma, University of Oxford (UK)
Olivier Martin, ICT Consulting (Switzerland)
Katsuichi Nakamura, Kyushu Institute of Technology (Japan)
Marcelo Pasin, INRIA, ENS-Lyon (France)
Nicholas Race, Lancaster University (UK)
Depei Qian, Beijing University of Aeronautics and Astronautics (China)
Dimitra Simeonidou, University of Essex (UK)
Burkhard Stiller, University of Zurich (Switzerland)
Zhili Sun, University of Surrey (UK)
Martin Swany, University of Delaware (USA)
Osamu Tatebe, Tsukuba University (Japan)
Dominique Verchere, Alcatel-Lucent (France)
Jun Wei, Institute of Software, Chinese Academy of Sciences (China)
Chan-Hyung Youn, ICU (Korea)
Wolfgang Ziegler, Fraunhofer-Gesellschaft (Germany)

**Wireless Grids Workshop Organizing Committee**

Frank H.P. Fitzek, Aalborg University (Denmark)
Marcos D. Katz, VTT (Finland)
Geng-Sheng Kuo, BUPT (China)
Jianwei Niu, BUAA (China)
Qi Zhang, DTU (Denmark)
# Table of Contents

A High Performance SOAP Engine for Grid Computing

*Ning Wang, Michael Welzl, and Liang Zhang*  
1

UDTv4: Improvements in Performance and Usability

*Yunhong Gu and Robert Grossman*  
9

The Measurement and Modeling of a P2P Streaming Video Service

*Peng Gao, Tao Liu, Yanming Chen, Xingyao Wu, Yehia El-khatib, and Christopher Edwards*  
24

SCE: Grid Environment for Scientific Computing

*Haili Xiao, Hong Wu, and Xuebin Chi*  
35

QoS Differentiated Adaptive Scheduled Optical Burst Switching for Grid Networks

*Oliver Yu and Huan Xu*  
43

Principles of Service Oriented Operating Systems

*Lutz Schubert and Alexander Kipp*  
56

Preliminary Resource Management for Dynamic Parallel Applications in the Grid

*Hao Liu, Amril Nazir, and Søren-Aksel Sørensen*  
70

Performance Evaluation of a SLA Negotiation Control Protocol for Grid Networks

*Igor Cergol, Vinod Mirchandani, and Dominique Verchere*  
81

Performance Assessment Architecture for Grid

*Jin Wu and Zhili Sun*  
89

Implementation and Evaluation of DSMIPv6 for MIPL

*Mingli Wang, Bo Hu, Shanzhi Chen, and Qinxue Sun*  
98

Grid Management: Data Model Definition for Trouble Ticket Normalization

*Dimitris Zisiadis, Spyros Kopsidas, Matina Tsavli, Leandros Tassiulas, Leonidas Georgiadis, Chrysostomos Tziouvaras, and Fotis Karayannis*  
105

Extension of Resource Management in SIP

*Franco Callegati and Aldo Campi*  
113

Economic Model for Consistency Management of Replicas in Data Grids with OptorSim Simulator

*Ghalem Belalem*  
121
<table>
<thead>
<tr>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A Video Broadcast Architecture with Server Placement Programming</td>
<td>130</td>
</tr>
<tr>
<td>Lei He, Xiangjie Ma, Weili Zhang, Yunfei Guo, and Wenbo Liu</td>
<td></td>
</tr>
<tr>
<td>VXDL: Virtual Resources and Interconnection Networks Description</td>
<td>138</td>
</tr>
<tr>
<td>Guilherme Piegas Koslovski, Pascale Vicat-Blanc Primet, and Andrea Schwertner Charão</td>
<td></td>
</tr>
<tr>
<td>Adding Node Absence Dynamics to Data Replication Strategies for Unreliable Wireless Grid Environments</td>
<td>155</td>
</tr>
<tr>
<td>Soomi Yang</td>
<td></td>
</tr>
<tr>
<td>Hop Optimization and Relay Node Selection in Multi-hop Wireless Ad-Hoc Networks</td>
<td>161</td>
</tr>
<tr>
<td>Xiaohua(Edward) Li</td>
<td></td>
</tr>
<tr>
<td>Localization Anomaly Detection in Wireless Sensor Networks for Non-flat Terrains</td>
<td>175</td>
</tr>
<tr>
<td>Sireesha Krupadanam and Huirong Fu</td>
<td></td>
</tr>
<tr>
<td>Network Coding Opportunities for Wireless Grids Formed by Mobile Devices</td>
<td>187</td>
</tr>
<tr>
<td>Karsten Fyhn Nielsen, Tatiana K. Madsen, and Frank H.P. Fitzek</td>
<td></td>
</tr>
<tr>
<td>Automatic Network Services Aligned with Grid Application Requirements in CARRIOCAS Project (Invited Paper)</td>
<td>196</td>
</tr>
<tr>
<td>Communication Contention Reduction in Joint Scheduling for Optical Grid Computing (Invited Paper)</td>
<td>206</td>
</tr>
<tr>
<td>Yaohui Jin, Yan Wang, Wei Guo, Weiqiang Sun, and Weisheng Hu</td>
<td></td>
</tr>
<tr>
<td>Experimental Demonstration of a Self-organized Architecture for Emerging Grid Computing Applications on OBS Testbed</td>
<td>215</td>
</tr>
<tr>
<td>Lei Liu, Xiaobin Hong, Jian Wu, and Jintong Lin</td>
<td></td>
</tr>
<tr>
<td>Joint Scheduling of Tasks and Communication in WDM Optical Networks for Supporting Grid Computing</td>
<td>223</td>
</tr>
<tr>
<td>Xubin Luo and Bin Wang</td>
<td></td>
</tr>
<tr>
<td>Manycast Service in Optical Burst/Packet Switched (OBS/OPS) Networks (Invited Paper)</td>
<td>231</td>
</tr>
<tr>
<td>Vinod M. Vokkarane and Balagangadhar G. Bathula</td>
<td></td>
</tr>
</tbody>
</table>
OBS/GMPLS Interworking Network with Scalable Resource Discovery for Global Grid Computing (Invited Paper) .......................... 243
J. Wu, L. Liu, X.B. Hong, and J.T. Lin

Providing QoS for Anycasting over Optical Burst Switched Grid Networks .......................................................... 251
Balagangadhar G. Bathula and Jaafar M.H. Elmirghani

A Paradigm for Reconfigurable Processing on Grid ................. 259
Mahmood Ahmadi and Stephan Wong

Author Index ........................................................................... 263
A High Performance SOAP Engine for Grid Computing

Ning Wang¹, Michael Welzl², and Liang Zhang¹

¹ Institute of Software, Chinese Academy of Sciences, Beijing, China
   wangning@otcaix.iscas.ac.cn
² Institute of Computer Science, University of Innsbruck, Austria
   michael.welzl@uibk.ac.at

Abstract. Web Service technology still has many defects that make its usage for Grid computing problematic, most notably the low performance of the SOAP engine. In this paper, we develop a novel SOAP engine called SOAPExpress, which adopts two key techniques for improving processing performance: SCTP data transport and dynamic early binding based data mapping. Experimental results show a significant and consistent performance improvement of SOAPExpress over Apache Axis.

Keywords: SOAP, SCTP, Web Service.

1 Introduction

The rapid development of Web Service technology in recent years has attracted much attention in the Grid computing community. The recently proposed Open Grid Service Architecture (OGSA) represents an evolution towards a Grid system architecture based on Web Service concepts and technologies. The new WS-Resource Framework (WSRF) proposed by Globus, IBM and HP provides a set of core Web Service specifications for OGSA. Taken together, and combined with WS-Notification (WSN), these specifications describe how to implement OGSA capabilities using Web Services.

The low performance of the Web Service engine (SOAP engine) is problematic in the Grid computing context. In this paper, we propose two techniques for improving Web Service processing performance and develop a novel SOAP engine called SOAPExpress. The two techniques are using SCTP as a transport protocol, and dynamic early binding based data mapping. We conduct experiments comparing the new SOAP engine with Apache Axis by using the standard WS Test suite¹. The experimental results show that, no matter what the Web Service call is, SOAPExpress is always more efficient than Apache Axis. In case of handling an echoList Web Service call, SOAPExpress can achieve a 56% reduction of the processing time.

After a review of related work, we provide an overview of SOAPExpress in section 3, with more details about the underlying key techniques following in section 4. We present a performance evaluation in section 5 and conclude.

2 Related Work

There have been several studies\cite{3,4,5} on the performance of the SOAP processing. These studies all agree that the XML based SOAP protocol incurs a substantial performance penalty in comparison with binary protocols. Davis conducts an experimental evaluation on the latency of various SOAP implementations, compared with other protocols such as Java RMI and CORBA\cite{4}. A conclusion is drawn that two reasons may cause the inefficiency of SOAP: one is about the multiple system calls to realize one logical message sending, and another is about XML parsing and formatting. A similar conclusion is drawn in\cite{3} by comparing SOAP with CORBA. Chiu et al. point out that the most critical bottleneck in using SOAP for scientific computing is the conversion between floating point numbers and their ASCII representations in\cite{3}.

Recently, various mechanisms have been utilized to optimize the deserialization and serialization between XML data and Java data. In\cite{1}, rather than re-serializing each message from scratch, a serialized XML message copy is cached in the senders stub, which is reused as a template for the next message with the same type. The approach in\cite{8} reuses the matching regions from the previously deserialized application objects, and only performs deserialization for a new region that has not been processed before; however, for large SOAP messages, especially for SOAP messages whose data always changes, the performance improvement of\cite{8} will be decreased. Also Java reflection is adopted by\cite{3} as a means to set and get new values. For large Java objects, especially deeply nested objects, this will negatively affect the performance.

The transport protocol is also a factor that can degrade the SOAP performance. The traditional HTTP and TCP communication protocols exhibit many defects when used for Web Services, including “Head-Of-Line blocking (HOL)” delay (to be explained in section\cite{11}), three-way handshake, ordered data delivery and half open connections\cite{2}. Some of these problems can be alleviated by using the SCTP protocol\cite{7} instead of TCP. While this benefit was previously shown for similar applications, most notably MPI (see chapter 5 of\cite{6} for a literature overview), to the best of our knowledge, using SCTP within a SOAP engine as presented in this paper is novel.

3 SOAPExpress Overview

As a lightweight Web Service container, SOAPExpress provides an integrated platform for developing, deploying, operating and managing Web Services, and fully reflects the important characteristics of the next generation SOAP engine including QoS and diverse message exchange patterns. SOAPExpress not only supports the core Web Service standards such as SOAP and WSDL, but also inherits the open and flexible design style of Web Service technology because of its architecture: SOAPExpress can easily support different Web Service standards such as WS-Addressing, WS-Security and WS-ReliableMessage. It can also be integrated with the major technology for enterprise applications such as EJB and
JMS to establish a more loosely coupled and flexible computing environment. To enable agile development of Web Services, we also provide development tools as plug-ins for the Eclipse platform.

The architecture of SOAPExpress consists of four parts as shown in Fig. 1:

- Transport protocol adaptor: supports client access to the system through a variety of underlying protocols such as HTTP, TCP and SCTP, and offers the system an abstract SOAP message receiving and sending primitive.
- SOAP message processing module: provides the effective and flexible SOAP message processing mechanism and is able to access the data in the SOAP message in three layers, namely byte stream, XML object and Java object.
- Execution controller: with a dynamic pipeline structure, controls the flow of SOAP message processing such as service identification, message addressing and message exchanging pattern management, and supports various QoS modules such as security and reliable messaging.
- Integrated service provider: provides an integrated framework to support different kinds of information sources such as plain Java objects and EJBs, and wraps them into Web Services in a convenient way.

4 Key Techniques

In this section, we will present the design details of the key techniques applied in SOAPExpress to improve its performance.

4.1 SCTP Transport

At the transport layer, we use the SCTP protocol [7] to speed up the execution of Web Service calls. This is done by exploiting two of its features: out-of-order delivery and multi-streaming. Out-of-order delivery eliminates the HOL delay of TCP: if, for example, packets 1, 2, 3, 4 are sent from A to B, and packet 1 is lost, packets 2, 3 and 4 arrive at the receiver before the retransmitted (and therefore delayed) packet 1. Then, even if the receiving application could already
use the data in packets 2, 3 and 4, it has no means to access it because the TCP semantics (in-order delivery of a consecutive data stream) prevent the protocol from handing over the content of these packets before the arrival of packet 1.

In a Grid, these packets could correspond with function calls, which, depending on the code, might need to be executed in sequence. If they do, the possibility of experiencing HOL blocking delay is inevitable — but if they don’t, the out-of-order delivery feature of SCTP can speed up the transfer in the presence of packet loss. Directly using the out-of-order delivery mechanism may not always be useful, as this would require each function call to be at most as large as one packet, thereby significantly limiting the number and types of parameters that could be embedded. We therefore used the multi-streaming feature, which bundles independent data streams together and allows out-of-order delivery only for packets from different streams. In our example, packets 1 and 3 could be associated with stream A, and packets 2 and 4 could be associated with stream B. The data of stream B could then be delivered in sequence before the arrival of packet 1, thereby speeding up the transfer.

For our implementation, we used a Java SCTP library which is based on the Linux kernel space implementation called “LKSCTP”\footnote{Java SCTP library by I. Skytte Joergensen: http://i1.dk/JavaSCTP/}. Since our goal was to enable the use of SCTP instead of TCP without requiring the programmer to carry out a major code change, we used Java’s inherent support for the factory pattern as a simple and efficient way to replace the TCP socket with an SCTP socket in an existing source code. All that is needed to automatically make all socket calls use SCTP instead of TCP is to call to the methods Socket.setSocketImplFactory and ServerSocket.setSocketFactory for the client and server side, respectively. In order to avoid bothering the programmer with the need to determine which SCTP stream a function call should be associated with, we automatically assign socket calls to streams in a round-robin fashion.

Clearly, it must be up to the programmer to decide whether function calls could be executed in parallel (in which case they would be associated with multiple streams) or not. To this end, we also provide two methods called StartChunk() and EndChunk(), respectively, which can be used to mark parts of data which must be consecutively delivered. All write() calls that are executed between StartChunk() and EndChunk() will cause data to be sent via the same stream.

### 4.2 Dynamic Early Binding Based Data Mapping

The purpose of the data mapping is to build a bridge between the platform independent SOAP messages and the platform dependent data such as Java objects. The indispensable elements of the data mapping include XML data definitions in an XML schema, data definitions in a specific platform, and the mapping rule between them. Before discussing our data mapping solution, let us first explain two pairs of concepts.

- Early binding and late binding: The differences between early binding and late binding focus on when to get the binding information and when to
use them, as illustrated in the Fig. 2. Here the binding information refers to mapping information between XML data and Java data. In early binding, all the binding information is retrieved before performing the binding, while in late binding, the binding is performed as soon as enough binding information is available.

- Dynamic binding and static binding: Here, dynamic binding refers to the binding mechanism which can add new XML-Java mapping pairs at run time. In contrast, static binding refers to a mechanism which can only add new mapping pairs at compilation time.

![Fig. 2. Early binding and late binding](image)

According to the above explanation, the existing data binding implementations can be classified into two schemes: dynamic late binding and static early binding. Dynamic late binding gets the binding information by Java reflection at run time, and then uses the binding information to carry on data binding between XML data and Java data. Dynamic late binding can dynamically add new XML-Java mapping pairs, and avoid generating assistant codes by using dynamic features of Java; however, this flexibility is achieved by sacrificing efficiency. Representatives of this scheme are Apache Axis and Castor. For example, Castor uses java reflection to instantiate the new class added to the XML-java pairs at the run time, and initialize it using the values in XML through method reflection. Static early binding generates Java template files which record the binding information before running, and then carries on the binding between XML data and Java data at runtime. Static early binding (as, e.g., in XMLBeans) improves the performance by avoiding the frequent use of Java reflection. However, new XML-Java mapping pairs cannot be added at runtime, which reduces the flexibility.

As illustrated in Fig. 3, we use a dynamic early binding scheme. This scheme can establish the mapping rules between the XML schema for some data type and the Java class for the same data type at compilation time. At run time, a Java template is generated based on the XML schema, the Java class and their mapping rules, which we call Data Mapping Template (DMT), by dynamic code generation techniques. The DMT is used to drive the data mapping procedure. Dynamic early binding avoids Java reflection so that the performance can be distinctly improved. Simultaneously, the DMT can be generated and managed at run time, which gives dynamic early binding the same flexibility as dynamic
late binding. Dynamic early binding combines the advantages of static early binding and dynamic late binding.

5 Performance Evaluation

We begin our performance evaluation with a study of the performance improvement from using SCTP with multi-streaming. We executed asynchronous Web Service calls with JAX-WS 2.0 to carry out a simple scalar product calculation, where sequential execution of the individual calls (multiplications) is not necessary. Three PCs were used: a server and client (identical AMD Athlon 64 X2 Dual-Core 4200 with 2.2 GHz), and a Linux router which interconnected them (a HP Evo W6000 2.4 GHz workstation). At the Linux router, we generated random packet loss with NistNet

Figure 3 shows the transfer time of these tests with various packet loss ratios. The results were taken as an average of running 100 tests with the same packet loss setting each. As could be expected from tests carried out in a controlled environment, there was no significant divergence between the results of these test runs. For each measurement, we sent 5000 Integer values to the Web Service, which sent 2500 results back to the client. Eventually, at the client, the sum was calculated to finally yield the scalar product.

Clearly, if SCTP is used as we intended (with unordered delivery between streams and 1000 streams), it outperforms TCP, in particular when the packet loss ratio gets high. SCTP with one stream and ordered behavior is only included in fig. 4 as a reference value — its performance is not as good as TCP’s because the TCP implementation is probably more efficient (TCP has evolved over many years and, other than our library, operates at the kernel level). Multiple streams and ordered transmission of packets between streams would theoretically be pointless; surprisingly, the result for this case is better than with only one stream. We believe that this is a peculiarity of the SCTP library that we used.

\[\text{http://snad.ncsl.nist.gov/nistnet/}\]
A High Performance SOAP Engine for Grid Computing

Fig. 4. Transfer time of TCP and SCTP using the Web Service

We then evaluated the performance of SOAPExpress as a whole, including SCTP and dynamic early binding based data mapping. We chose the WS Test 1.0 suite to test the time spent on each stage in the SOAP message processing. Several kinds of test cases were carried out, each designed to measure the performance of a different type of Web Service calls:

- echoVoid: send/receive a message with empty body.
- echoStruct: send/receive an array of size 20, with each entry being a complex data type composed of an integer, a floating point number and a string.
- echoList: send/receive a linked list of size 20, with each entry being the same complex data type defined in echoStruct.

The experimental settings were: CPU: Pentium-4 2.40 GHz; Memory: 1 GB; OS: Ubuntu Linux 8.04; JVM: Sun JRE 6; Web Container: Apache Tomcat 6.0. The Web Service client performed each Web Service call 10,000 times, and the workload was 5 calls per second.

The benchmark we have chosen is Apache Axis 1.2. Fig. 4 shows the experimental results. For echoStruct and echoList, the XML payload was about 4KB. The measure started from receiving a request and ended with returning the

Fig. 5. Performance comparison among different types of Web Service calls
response. We observed that for echoVoid, the processing time is very close between the two SOAP engines, since echoVoid has no business logic and just returns the SOAP message with an empty body. For echoStruct, the processing time of SOAPExpress is about 46% of Apache Axis, and for echoList, the proportion reduces to about 44%. This is a very sound overall performance improvement of SOAPExpress over Apache Axis.

6 Conclusion

In this paper, we presented the SOAPExpress engine for Grid computing. It uses two key techniques for reducing the Web Service processing time: the SCTP transport protocol and dynamic early binding based data mapping. Experiments were conducted to compare its performance with Apache Axis by using the standard WS Test suite. Our experimental results have shown that, no matter what the Web Service call is, SOAPExpress is more efficient than Apache Axis.

Acknowledgments

We thank Christoph Sereinig for his contributions. This work is partially supported by the FP6-IST-045256 project EC-GIN (http://www.ec-gin.eu).

References

UDTv4: Improvements in Performance and Usability

Yunhong Gu and Robert Grossman
National Center for Data Mining, University of Illinois at Chicago

Abstract. This paper presents UDT version 4 (UDTv4), the fourth generation of the UDT high performance data transfer protocol. The focus of the paper is on the new features introduced in version 4 during the past two years to improve the performance and usability of the protocol.

UDTv4 introduces a new three-layer protocol architecture (connection-flow-multiplexer) for enhanced congestion control and resource management. The new design allows protocol parameters to be shared by parallel connections and to be reused by future connections. This improves the congestion control and reduces the connection setup time. Meanwhile, UDTv4 also provide better usability by supporting a broader variety of network environments and use scenarios.

1 Introduction

During the last decade there has been a marked boom in Internet applications, enabled by the rapid growth of raw network bandwidth. Examples of new applications include P2P file sharing, streaming multimedia, and grid/cloud computing. These applications vary greatly in traffic and connection characteristics. However, most of them still use TCP for data transfer. This is partly due to the fact that TCP is well established and contributes to the stability of the Internet.

TCP was designed as a general-purpose protocol and was first introduced three decades ago. It is not surprising that certain requirements from new applications cannot be perfectly addressed by TCP. Network researchers have proposed many changes to TCP to address those emerging problems and requirements (SACK, ECN, etc.) [6]. The new techniques are carefully studied and deployed, albeit slowly. For example, TCP's inefficiency problem in high bandwidth-delay product (BDP) networks was observed almost a decade ago yet it is only recently that several new high speed TCP variants were deployed: (CUBIC on Linux [13] and Compound TCP on Windows Vista [17]). Furthermore, because new TCP algorithms have to be compatible with the TCP standard, improvements to TCP are limited.

New transport protocols, DCCP [12] and SCTP [16], have also been proposed. However, it may take years for these new protocols to be widely deployed and used by applications (considering the example of IPv6). Moreover, both DCCP and SCTP are designed for specific groups of applications. New applications and requirements will continue to emerge and it is not a scalable solution to design a new transport layer protocol every few years. It is necessary to have a flexible protocol that provides basic functions and allows applications to define their own data processing. This is what UDP was designed for.
In fact, UDP has been used in many applications (e.g., Skype) but it is usually customized independently for each application. RTP [15] is a good example and it is a great success in supporting multimedia applications. However, there are few general-purpose UDP-based protocols that application developers can use directly or customize easily.

UDT, or UDP-based data transfer protocol, is an application level general-purpose transport protocol on top of UDP [8]. UDT address a large portion of the requirements from the new applications by seamlessly integrating many modern protocol design and implementation techniques at the application level.

The protocol was originally designed for transferring large scientific data over high-speed wide area networks and it has been successful in many research projects. For example, UDT has been used to distribute the 13TB SDSS astronomy data release to global astronomers [9].

UDT has been an open source project since 2001 and the first production release was made in 2004. While it was originally designed for big scientific data sets, the UDT library has been used in many other situations, either with its stock form or in a modified form. A great deal of user feedback has been received. The new version (UDTv4) released in 2007 introduces significant changes and supports better performance and usability.

- UDTv4 uses a three-layer architecture to enhance congestion control and reduce connection setup time by sharing control parameters among parallel connections and by using historical data.
- UDTv4 introduces new techniques in both protocol design and implementation to support better scalability, hence it can be used in a larger variety of use scenarios.

This paper describes these new features of UDTv4. Section 2 explains the protocol design. Section 3 describes several key implementation techniques. Section 5 presents the evaluation. Section 6 discusses the related work. Section 7 concludes the paper. Throughout the rest of the paper, we use UDT to refer the most recent version, UDTv4, unless otherwise explicitly stated.

2 Protocol Design

2.1 Protocol Overview

UDT is a connection-oriented, duplex, and unicast protocol. There are 3 logical layers in design: UDT connection, UDT flow, and UDP multiplexer (Figure 1).

A UDT connection is set up between a pair of UDT sockets as a distinct data transfer entity to applications. It can provide either reliable data streaming services or partial reliable messaging services, but not both for the same socket.

A UDT flow is a logical data transfer channel between two UDP addresses (IP and port) with a unique congestion control algorithm. That is, a UDT flow is composed of five elements (source IP, source UDP port, destination IP, destination UDP port, and congestion control algorithm). The UDT flow is transparent to applications.
One or more UDT connections are associated with one UDT flow, if the UDT connections share the same five elements described above. Every connection must be associated with one and only one flow. In other words, UDT connections sharing the same five elements are multiplexed over a single UDT flow.

A UDT flow provides reliability control as it multiplexes individual packets from UDT connections, while UDT connections provide data semantics (streaming or messaging) management. Different types of UDT connections (streaming or messaging) can be associated with the same UDT flow.

Congestion control is also applied to the UDT flow, rather than the connections. Therefore, all connections in one flow share the same congestion control process. Flow control, however, is applied to each connection.

Multiple UDT flows can share a single UDP socket/port and a UDP multiplexer is used to send and dispatch packets for different UDT flows. The UDP multiplexer is also transparent to applications.

### 2.2 UDP Multiplexing

Multiple UDT flows can bind to a single UDP port and each packet is differentiated by the destination (UDT) socket ID carried in the packet header. The UDP multiplexing method helps to traverse firewalls and alleviates the system limitation on the port number space. The number of TCP ports is limited to 65536. In contrast, UDT can support up to $2^{32}$ connections at the same time.

UDP multiplexing also helps firewall traversing. By opening one UDP port, a host can open virtually an unlimited number of UDT connections to the outside.

### 2.3 Flow Management

UDT multiplexes multiple connections into one single UDT flow, if the connections share the same attributes of source IP, source UDP port, destination IP, destination UDP port, and congestion control algorithm.

This single flow for multiple connections helps to reduce control traffic, but more importantly, it uses a single congestion control for all connections sharing the same end points. This removes the unfairness by using parallel flows and in most situations
it improves throughput because connections in a single flow coordinate with each other rather than compete with each other.

As shown in Figure 2, the flow maintains all activities required for a regular data transfer connection, whereas the UDT connection is only responsible for the application interface (connection maintenance and data semantics).

At the sender side, the UDT flow reads packets from each associated connection in a round robin manner, assigns each packet the flow sequence numbers and sends them out.

### 2.4 Connection Record Index/Cache

When a new connection is requested, UDT needs to look up whether there is already a flow existing between the same peers. A connection record index (Figure 3) is used for this purpose.

The index is sorted by the peer IP addresses. Each entry records the information between the local host and the peer address, including but not limited to RTT, path MTU, and estimated bandwidth. Each entry may contain multiple sub-entries by different ports, followed by multiple flows differentiated by congestion control (CC).

The connection record index caches the IP information (RTT, MTU, estimated bandwidth, etc.) even if the connection and flow is closed, in which case there is no port associated with the IP entry. This information can be used when a new connection is set up. Its RTT value can be initialized with a previously recorded value; otherwise it would take several ACKs to get an accurate value for the RTT. If
path MTU discovery is used, the MTU information can also be initialized with a historical value.

The index entry without an active flow will be removed when the maximum length of the index has been reached, and the oldest entry will be removed first.

Although the cache may be removed very quickly on a busy server (e.g., a web server), the client side may contain the same cache and pass the values to the server. For example, a client that frequently visits a web server may keep the link information between the client and the server, while the server may have already removed it.

2.5 Garbage Collection

When a UDT socket is closed (either by the application or because of a broken connection), it is not removed immediately. Instead, it is tagged as having closed status. A garbage collection thread will periodically scan the closed sockets and remove the sockets when no API is accessing the socket.

Without garbage collection, UDT would have needed stronger synchronization protection on its APIs, which increases implementation complexity and adds some slight overhead for the additional synchronization mechanism.

In addition, because of the delayed removal, a new socket can reuse a closed socket and the related UDP multiplexer when possible, thus it improves connection setup efficiency.

Garbage collection also checks the buffer usage and decreases the size of the system allocated buffer if necessary. If during the last 60 seconds, less that 50% of the buffer is used, the buffer will be reduced to half (a minimum size limit, 32 packets, is used so that the buffer size will not be decreased to a meaningless 1-byte).

3 Implementation

UDT is implemented as an open source project and is available for download from SourceForge.net. The UDT library has been used in both research projects and commercial products. So far 18,000 copies have been downloaded, excluding direct checkout from the CVS and redistribution from other websites.

The UDT implementation is available on both POSIX and Windows systems and it is thoroughly tested on Linux 2.4, 2.6, and Windows XP. The code is written in C++ with API wrappers for other languages available.

The latest stable version of the UDT library (version 4.2) consists of approximately 11,500 lines of C++ code, including about 4000 semicolons and about 20% of the code is comments.

3.1 Software Architecture

Figure 4 shows the software architecture of the UDT implementation. A global UDT API module dispatches requests from applications to a specific UDT socket. Data transfer for the UDT socket is managed by a UDT flow, while the UDT flow communicates via a UDP multiplexer. One UDP multiplexer can support multiple UDT flows, and one UDT flow can support multiple UDT sockets. Finally, both the
buffer management module and the garbage collection module work at global space to support the resource management.

Figure 5 shows the data flow in a single UDT connection. The UDT flow moves data packets from the socket buffer to its own sending buffer and sends the data out via the UDP multiplexer. The control information is exchanged on both directions of the data flow. At the sender side, the UDP multiplexer receives the control information (ACK, NAK, etc.) from the receiver and dispatches the control information to the
corresponding UDT flow or connection. Lost lists are used at both sides to record the lost packets. Lost lists work at flow level and only record flow sequence numbers. Flow control is applied to a UDT socket, while congestion control and reliability control are applied to the UDT flow.

### 3.2 UDP Multiplexer and Queue Management

The UDP multiplexer maintains a sending queue and a receiving queue. The queue manages a set of UDT flows to send or receive packets via the associated UDP port.

The sending queue contains a set of UDT flows that has data to send out. If rate based control is used, the flows are scheduled according to the next packet sending time; if pure window-based control is used, the flows are scheduled according to a round robin scheme.

The sending queue checks the system time and when it is time to send out the first packet, it removes the first flow on the queue and sends out its packet. If there are more packets to be sent for the particular flow, the flow will be inserted into the queue again according to the next packet sending time by rate/congestion/flow control.

The sending queue uses a heap structure to maintain the flows. With the heap structure, each send or insert action takes at most \( \log_2(n) \) steps, where \( n \) is the total number of flows in the queue. The heap structure guarantees that the sender can find the flow instance with the smallest next scheduled packet sending time; however, it is not necessary to have all the flows sorted by the next scheduled time.

The job of the receiving queue is much simpler. It checks the timing events (retransmission timer, keep-alive, timer-based ACK, etc.) for each flow associated with the UDP multiplexer. Every fixed time interval (0.1 second), flows are checked in a round robin manner. However, if a packet arrived for a particular flow, the timers will be checked for the flow and the flow is moved to the end of the queue for the next round of check.

The receiving queue uses a double linked list to store the flows and each operation takes \( O(1) \) time.

The receiving side of the UDP multiplexer also maintains a hash table for the associated UDT connections, so that when a packet arrives, the multiplexer can quickly look up the corresponding connection to process the packet. Note that the flow processing handler can be looked up via the socket instance.

### 3.3 Connection and Flow Management

In the UDT implementation, a flow is a special connection that contains pointers to all connections within the same flow, including itself.

The first connection of the flow is set up by the normal 3-way handshake process. More connections are set up by a simplified 2-way handshake as it joins an existing flow. The first connection automatically becomes the flow and manages all the connections. If the current "flow" connection is closed or leaves (because of IP address change), another connection will become the flow and related flow information will be moved to the new flow from the old one.

The flow maintains a separate sending buffer in addition to the connections' sending buffers. In an ideal world, the flow should read packets from each connection
in a round robin fashion. However, in this way the flow would either need to keep track of the source of each packet or copy the packet into its own buffer, because each ACK or NAK processing needs to locate the original packet.

In the current implementation, the socket sending buffer is organized as a link of multiple 32-packet blocks. The UDT flow reads one 32-packet block from each connection in round robin fashion, removes the block from the socket's sending buffer, and links the block to its own (flow) sending buffer. Note that there may be less than 32 packets in the block if there is not enough data to be sent for a particular connection.

Flow control is enforced at the socket level. The UDT send call will be blocked if either the sender buffer limit or the receiver buffer limit is full. This guarantees that data in the flow sending buffer is not limited by flow control.

By using this strategy, the flow simply applies ACKs and NAKs to its own buffer and avoids memory copies between flow and connections or a data structure to map flow sequence number to connection sequence number. In the latter case, UDT would also need to check every single packet being acknowledged, because they may belong to different connections and may not be continuous.

At the receiver side, all connections have their own receiver buffer for application data reading. However, only the flow maintains a loss list to recover packet losses.

**Rendezvous connection setup.** In addition to the regular client/server mode, UDT provides a method for rendezvous connection method. Both peers can connect to each other at (approximately) the same time, provided that they know the peer's address beforehand (e.g., via a 3rd known server).

### 3.4 Performance Considerations

**Multi-core processing.** The UDT implementation uses multiple threads to explore the multi-core ability of modern processors. Network bandwidth increases faster than CPU speed, and a single core of today's processors is barely enough to saturate 10Gb/s.

One single UDT connection can use 2 cores (sending and receiving) per data traffic direction on each side. Meanwhile, each UDP multiplexer has its own sending thread and receiving thread. Therefore, users can start more UDT threads by binding UDT sockets to different UDP ports, thus more UDP multiplexers will be started and each multiplexer will start their own packet processing threads.

**New select API.** UDT provides a new version of the select API, in which the result socket descriptor set is an independent output, rather than overwriting the input directly. The BSD style select API is inefficient for large numbers of sockets, because the input is modified and applications have to reinitialize the input each time. In addition, UDT provides a way to iterate the result set; in contrast, for the BSD socket API, applications have to test each socket against the result set.

**New sendfile/recvfile API.** UDT provides both sendfile and recvfile APIs to reduce one memory copy by exchanging data between the UDT buffer and application file directly. These two APIs also simplify application development in certain cases.

It is important to mention that file transfer can operate under both streaming mode and messaging mode. However, messaging mode is more efficient in this case,
because \textit{recvfile} does not require continuous data block receiving and therefore in messaging mode data blocks can be read into files out of order without the "head of line" blocking problem. This is especially useful when the packet loss rate is high.

**Buffer auto-sizing.** All UDT connections/flows share the same buffer space, which increases when necessary. The UDT socket buffer size is only an upper limit and it does not allocate the buffer until it has to.

UDT automatically increases the socket buffer size limit to 2*BDP, if the default or user-specified buffer size is less than this value. However, if the default or user-specified value is greater than this value, UDT will not decrease the buffer size. The bandwidth value (B in BDP) is estimated by the maximum packet arriving rate at the receiver side. The garbage collection thread may decrease the system buffers when it detects that only less than half of the buffers are used.

### 4 Evaluation

This section evaluates UDT's scalability, performance, and usability. UDT provides superior usability over TCP and although it is at the application level, its implementation efficiency is comparable to the highly optimized Linux TCP implementation in kernel space. More importantly, UDT effectively addresses many application requirements and fills a blank left by transport layer protocols.

#### 4.1 Performance Characteristics

This section summarizes the performance characteristics of UDT, in particular, its scalability.

**Packet header size.** UDT consumes 24 bytes (16-byte UDT + 8-byte UDP) for data packet headers. In contrast, TCP uses a 20-byte packet header, SCTP uses a 28-byte packet header, and DCCP uses 12 bytes without reliability.

**Control traffic per flow.** UDT sends one ACK per 0.01 second when there is data traffic. This can be overridden by a user-defined congestion control algorithm, if more ACKs are necessary. However, the user-defined ACKs will be lightweight ACKs and consumes less bandwidth and CPU [8]. ACK2 packet is generated occasionally, at a decreased frequency (up to 1 ACK2 per second). In contrast, TCP implementations usually send one ACK every one or two segments.

In addition, UDT may also send NAKs, message drop request, or keep-alive packets when necessary, but these packets are much less frequent than ACK and ACK2.

**Limit on number of connections.** The maximum number of flows and connections supported by UDT is virtually only limited by system resources \((2^{32})\).

**Multi-threading.** UDT starts 2 threads per UDP port, in addition to the application thread. Users can control the number of data processing threads by using a different number of UDP ports.

**Summary of data structures.** At the UDP multiplexer level, UDT maintains the sending queue and receiving queue. The sending queue costs O(log\(_2n\)) time to insert
or remove a flow, where \( n \) is the total number of flows. The receiving queue checks timers of each UDT flow every 0.1 second, but it is self clocked by the arrival of packets. Each check costs \( O(1) \) time. Finally, the hash table used for the UDP multiplexer to locate a socket costs \( O(1) \) look up time.

The UDT loss list is based on congestion events, and each scan time is proportional to the number of congestion events, rather than the number of lost packets [8].

### 4.2 Implementation Efficiency

UDT’s implementation performance has been extensively tuned. This sub-section lists the CPU usage for one or more data flows between two local directly connected identical Linux servers. The server runs Debian Linux (kernel 2.6.18) on dual AMD Opteron Dual Core 3.0GHz processors, 4 GB memory, and 10GE MyriNet NIC. All system parameters are left as default except that the MTU is set to 9000 bytes. No TCP or UDP offload is enabled.

Figure 6 shows the CPU usage of a single TCP, UDP and UDT flow (with or without memory copy avoidance). The total CPU capacity is 400%, because there are 4 cores. Because each flow has a different throughput (varies between 5.4Gb/s TCP and 7.5Gb/s UDT with memory copy avoidance), the values listed in Figure 6 are CPU usage per Gb/s throughput.

According to Figure 6, UDT with memory copy avoidance costs similar CPU as UDP and less CPU time than TCP. UDT without memory copy avoidance costs approximately double CPU time of that in the other three situations.

In the case of a single UDT flow without memory copy avoidance, at 7.4Gb/s, the CPU usage of the UDT thread and the application thread at the sender side cost 99% and 40%, respectively (per thread CPU time not shown in Figure 6); the UDT thread and the application thread at the receiver thread cost 90% and 36%, respectively.

Fig. 6. CPU Usage of Single Data Flow