A Network Substrate for Peer-to-Peer Grid Computing beyond Embarrassingly Parallel Applications

Sven Schulz* Wolfgang Blochinger Mathias Poths*
University of Tübingen, Symbolic Computation Group
Sand14, D-72076 Tübingen, Germany
{schulzs, blochinger, poths}@informatik.uni-tuebingen.de

Abstract

Embarrassingly parallel Grid applications require no inter-worker interaction. Thus, the underlying communication infrastructure is based on the simple but well understood client/server execution model. This is no longer sufficient if more demanding parallel applications that require true Peer-to-Peer interaction are to be deployed on Grids. In this paper, we propose the industrial-strength eXtensible Messaging and Presence Protocol *(XMPP) as a network substrate for these non-trivial Peer-to-Peer Grid Computing applications.*

1. Introduction

Peer-to-Peer (P2P) Grid systems harness underutilized resources of (geographically distributed) desktop computers to tackle computationally challenging problems. Due to a remarkable cost-benefit ratio, P2P Desktop Grid Computing has become a promising alternative to classical Grids (e.g. based on the Globus toolkit) for certain kinds of applications.

Both approaches to Grid Computing ultimately pursue the same goal: aggregation of resources beyond local administrative domains. Thus P2P Grid Computing can be regarded as a special discipline of Grid Computing. However, there are also significant differences. Particularly, resources exhibit a much higher degree of volatility and heterogeneity. Delivering sustained computing power in such a dynamic and diverse environment is highly challenging, raising a plethora of research challenges. In contrast to other applications of the P2P paradigm, like file sharing [18], that have been shown to exhibit utmost scalability, Desktop Grid Computing applications are subject to a trade-off between scalability and the potential of the parallel programming model. While embarrassingly parallel Desktop Grid applications [6] require no inter-worker interaction and therefore scale to hundreds of thousands of hosts, the scalability of more complex applications is limited to networks of rather hundreds than thousands of hosts. Our previous work focused on software architectures [22, 21] for and applications [10, 9] of P2P Desktop Grids.

In this work, we deal with the underlying communication infrastructure for P2P Grid Computing systems with support for non-trivial parallel applications, often called the *(network) substrate*. Such a substrate provides support for efficient P2P interaction and serves as a basic building block for higher-level protocols, services (e.g. discovery and coordination), and applications.

The key contributions of our paper are as follows: First, we specify functional and non-functional requirements defining a substrate concept suitable for P2P Grid Computing. Second, we demonstrate that the abstractions and the infrastructure of the *eXtensible Messaging and Presence Protocol* (XMPP) are basically well-suited to implement such a substrate. Third, we identify the limitations of XMPP in the light of P2P Grid Computing and show pertinent extensions and improvements which leverage the well-defined protocol extension mechanism of XMPP. Finally, we provide detailed benchmark results to underpin our approach in general and the effectiveness of our improvements in particular.

2. Requirements

In this section, we identify *functional* and *non-functional requirements* for a Peer-to-Peer Grid Computing substrate (below referred to as P2P-GCS). We define the term *functional requirement* as the set of operations a P2P-GCS must provide to support the P2P interaction model within a Desktop Grid environment. Non-functional requirements describe the qualities of the operations provided by the P2P-GCS. Our definition encompasses basic operations required to build higher level services, in particular structured as

*Supported by Deutsche Forschungsgemeinschaft (DFG) under grant BL 941/1-1 and BL 941/1-2.

DRAFT

well as unstructured approaches to resource discovery. We derived and verified the requirements by integrating our implementation as the default substrate into our COHESION P2P Desktop Grid Computing platform [22] (see also `http://www.cohesion.de`).

2.1. Functional Requirements

(F1) Group Abstraction. Knowledge of other peers is the most fundamental requirement for a distributed system adhering to the P2P interaction model. In contrast to traditional parallel systems the set of nodes within the system is not static. Membership is in a constant state of flux, where nodes join and leave in an unpredictable, often uncorrelated manner. This phenomenon is called *volatility* or *churn* [15]. In order to interact, a node must be able to track the changes in membership at least for a subset of all present nodes. Since the relations of a node to others may be manifold and may change over time, we also require a group abstraction, where groups are first class objects, i.e. a node may instantiate and destroy as many groups as required.
(F2) Communication. To support interaction among peers a P2P-GCS must at least provide point-to-point communication. Additionally, many distributed algorithms are based on a broadcast communication primitive, that allows for delivering a piece of information to all nodes in the system. Since we also require a group abstraction, broadcast becomes groupcast communication, where information is delivered to all members of a group.

2.2. Non-Functional Requirements

(N1) Performance. Achieving high parallel efficiency is a major goal in P2P Grid computing. Thus, a P2P-GCS must provide low-latency and bandwidth economical communication. It should also deliver changes in membership views promptly enabling efficient use of resources.
(N2) Scalability. Todays largest Desktop Grid Computing systems [7] are comprised of up to hundreds of thousands of nodes. This paramount scalability is possible since application support is limited to embarrassingly parallel applications within a client/server interaction model. However, if support for more non-trivial parallelism [22] is required, P2P interaction becomes mandatory. Scalability in such systems is achieved by distributing state over the participating peers. Unfortunately, this decentralization necessitates communication for synchronization and coordination among the peers. Hence, there is a trade-off between scalability and performance. In contrast to other P2P applications, where absolute performance is of less importance, the performance requirement cannot be neglected for P2P Desktop Grids, since it is crucial for achieving adequate parallel efficiency.
(N3) Connectivity. As opposed to traditional parallel systems connectivity in P2P Grids is typically limited due to NAT devices and restrictive firewalls. A common solution to this problem is *relaying*, where a mediator node, that is reachable by both parties, accepts messages from a node and forwards them to the respective other node. However, this introduces a bottleneck. Fortunately, more efficient solutions like NAT hole punching have been developed recently. A P2P-GCS should be able to bridge network segmentations in an efficient way and thus provide universal connectivity.
(N4) Security. Since large-scale P2P Grids typically span more than a single administrative domain, a P2P-GCS must undertake measures to keep sensitive data private and to protect the system state from malicious participants. This includes securing communication as well as restricting access to groups.

3. XMPP Overview

The *Extensible Messaging and Presence Protocol* (XMPP) is an open, XML-based protocol for real-time communication that has been formalized by the *Internet Engineering Task Force* (IETF). As XMPP is modular, it can be easily extended to adapt to use cases not covered by the core specifications published as RFC 3920 and RFC 3921. Extensions are managed by the *XMPP Software Foundation* [5] as publicly available *XMPP Extension Protocols* (XEP). Historically, XMPP has been used for instant messaging (Jabber IM) and presence information. Presence is the IM term for information whether a user is available or not. Due to its extensibility, the scope of XMPP has grown significantly since it's invention. All kinds of applications based on real-time message exchange including media negotiation, whiteboarding, collaboration, content syndication, and generalized XML routing have been built on top of XMPP. Today, XMPP-based software is deployed on thousands of servers across the Internet.
Network architecture. The XMPP network is organized in a way that resembles the email network. Every user has a unique *Jabber ID* (JID). The JID consists of a user name and a DNS server name separated by an *at* sign, such as `foo@cohesion.de`. *XMPP entities*, i.e. clients and servers, communicate by exchanging XMPP messages called *stanzas*.

The XMPP network is decentralized and uses a simple message routing mechanism. If an XMPP server (`first.cohesion.de`) receives a message addressed to `bar@second.cohesion.de` from a locally connected user `foo`, the message is forwarded to the XMPP server at `second.cohesion.de`. For that purpose the server typically maintains a cache of inter-server connections. Servers that cooperate by routing messages to each other are called

DRAFT

federated in XMPP jargon. XMPP users can interact seamlessly with users located on other networks (with a different protocol stack) through special components called *gateways*.

Connectivity. Today many clients are behind restrictive firewalls that allow outgoing traffic only on the HTTP port. XMPP defines an HTTP binding that can be used by these clients to connect to the server using a long-lived HTTP connection. The HTTP binding adheres to a push-model to deliver messages, i.e. they are delivered as soon as they are sent. In contrast to polling, where many polls return no new messages, this model is more efficient. As XMPP employs the client/server model, NAT traversal is no problem.

Security. XMPP provides several levels of security at the protocol level. Spoofing is impeded by forcing clients (for client-to-server connections) and servers (for server-to-server connections) to authenticate to their host server. Server dialback and whitelisting are additional security measures used to control which server-to-server connections are allowed. Since both, the connection establishment phase and the communication phase of the protocol are secured by SASL and TLS, client/server communication in XMPP is inherently secure. As stanzas are potentially routed over intermediary servers that may belong to third-parties, XMPP provides end-to-end signing and object encryption [19] to prevent rogue servers from spying on communications.

Multi User Chats. *Multi-User Chat* (MUC), defined in XEP-0045, extends the XMPP protocol to enable several clients to communicate in a many-to-many fashion. The scope of such a conversation is defined by *rooms*. A room is a set of $(JID, role)$ pairs maintained by the XMPP server. Depending on its role, each occupant, identified by its JID, has certain rights within the room (e.g. kick or invite users). The hosting XMPP server propagates changes in room membership to all occupants using *presence* stanzas. Thus, clients can keep their membership lists up-to-date. Occupants within a MUC room can send *message* stanzas addressed to the room. Such messages are delivered to all occupants by the XMPP server. One-to-one communication is supported by XMPP private chats. As described in the following section, we use MUC rooms to implement both the group abstraction and the groupcast communication primitive.

Implementations. With EJABBERD [1] and OPENFIRE [3] there are two industrial strength XMPP server implementations available, that are capable of serving large networks handling thousands of concurrent connections. EJABBERD provides superior performance and fault-tolerance through clustering support. Unfortunately, it is written in Erlang, for which to our knowledge there is still no Fast Infoset (see Section 7) implementation available, which we use to improve XMPP protocol efficiency. OPENFIRE is written in Java™for which a Fast Infoset implementation exists [2].

Figure 1. Join operation

```
procedure JOIN(name, T)
    joined ← false
    while not joined do
        roominfo ← GET-ROOM-INFO(name)
        if roominfo ≠ NULL then
            joined ← JOIN-ROOM(name)
            if not joined then
                SLEEP(T)
            end if
            continue
        else
            joined ← CREATE-ROOM(name)
            if joined then
                CONFIGURE-ROOM(name)
            end if
        end if
    end while
end procedure
```

Although there are tens of client libraries available some of them implementing a large subset of XEPs, most of them lack support for Jingle and it's derivative XEPs (see Section 5). One of the most feature-rich implementations with Jingle support is SMACK [3]. Like OPENFIRE, SMACK is written in Java™.

We use OPENFIRE and SMACK as the software stack in our experiments and as a basis to implement our optimizations.

4. An XMPP Based Substrate

In this section we describe, how the elements of the XMPP protocol can be used to design a substrate for P2P Grid Computing that satisfies the functional and non-functional requirements identified in Section 2.

4.1. Implementation of Functional Requirements

The Group abstraction (**F1**) is implemented using XMPP MUCs. An XMPP server can host any number of MUCs. Thus peers are free to create as many groups as required. Although, the basic group operations can be implemented in a straightforward way, things are complicated by the fact that those operations are performed concurrently by multiple users. The process of joining a group consists of the following steps (see Figure 1): First, we check, whether information about the room is available. If so, we try to join the MUC. If we succeed, we are done. If not, the MUC has already been created by another node but is not yet configured (i.e. to be publicly available and to allow for unlimited occupants). Thus, we sleep for a short period of time T, giving the server a chance to complete the configuration process, before we retry to join the MUC. If there was no room information available, the room has not been created yet. Thus, we try to create the room. If successful, we configure the room to be public (i.e. everyone can join), and to

DRAFT

accept an unlimited number of occupants. Otherwise room creation fails, because another user has concurrently created the room. Thus, we try to join the room. Leaving a group is trivial, since we simply have to leave the MUC. Client-side membership lists are maintained by tracing the presence stanzas delivered by the XMPP server and updating the list of available nodes accordingly.

The required unicast and groupcast communication primitives (**F2**) can be easily mapped to the functionality provided by MUCs. While the former is implemented using private chats, the latter uses the fact that any occupant of a MUC room, can send a message to the room, i.e. to all room occupants.

4.2. Implementation of Non-Functional Requirements

XMPP already satisfies some of the non-functional requirements. It allows for universal connectivity (**N3**) as communication is client-initiated and an HTTP binding exists to tunnel restrictive server-side firewalls. With multiple security measures at the protocol level it fulfills our security requirements (**N4**).

However, XMPP has three major drawbacks concerning our non-functional requirements: First, it implements no P2P interaction model: Even those messages that could be exchanged directly between two clients are relayed by the XMPP server. Second, the MUC protocol used to implement our group abstraction maintains complete membership information at all nodes, resulting in maintenance costs that grow quadratically with group size. Third, XMPP is an XML protocol that is very bloated and thus unnecessarily wastes bandwidth and processing power. Hence, the server definitely becomes a performance bottleneck, when groups grow large and/or a large number of messages have to be relayed. Without further optimizations our substrate would fail to satisfy the non-functional requirements for performance (**N1**) and scalability (**N2**).

Fortunately, there are pertinent solutions available for all three problems: With *Jingle*, which is explained in detail in Section 5, there is a XEP available that allows for negotiating P2P links between users. These links can be used to deliver unicast messages between clients, thus taking considerable parts of the load off the XMPP server. In Section 6, we describe how the XMPP MUC protocol can be extended to support partial membership views of configurable size resulting in maintenance costs that are logarithmic with respect to group size. Finally, we describe how *Fast Infoset*, a binary encoding of XML, can be integrated into an XMPP server without sacrificing scalability in Section 7. As will be substantiated in Section 8, this optimizations significantly improve the performance (**N1**) and scalability (**N2**) of our substrate.

5. Efficient P2P Interaction

XMPP servers act as relays for exchanging messages between connected clients. While such a connection model enables hosts behind firewalls or NAT devices to communicate with each other, the indirection over one or more servers limits scalability and performance unnecessarily, when hosts would be able to communicate directly. Hence, a modification of the XMPP private chat subsystem to enable P2P communication promises to increase overall system-wide message throughput (by eliminating the performance bottlenecks induced by XMPP servers) and to decrease message latency (by reducing the number of necessary hops from three to one). With this addition, our infrastructure becomes very similar to the infrastructure of the JXTA [24] network (see Figure 2): The XMPP server is, like the *rendezvous peer* in JXTA, responsible for delivering groupcast messages to all connected users. Unicast messages are delivered over direct connections between clients. If no direct connection is possible due to NAT devices or firewalls, the XMPP server delivers the message conventionally. This is similar to the responsibility of *relay peers* in the JXTA network architecture.

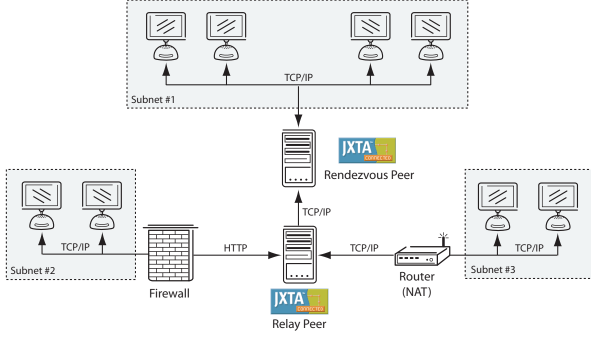
5.1. Jingle

Jingle is an XMPP extension (XEP-0166) developed to enable P2P sessions between XMPP entities. It is a pure signaling protocol, i.e. Jingle controls the connection negotiation process over the XMPP channel, while P2P interaction is accomplished *out-of-band* using custom communication technologies like the *Real-time Transport Protocol* (RTP), the *User Datagram Protocol* (UDP), or the *Interactive Connectivity Establishment* (ICE) protocol. Jingle is primarily targeted to support media exchange applications like voice or video chats. However, due to its modular design Jingle can be easily extended to support other session types and transport mechanisms.

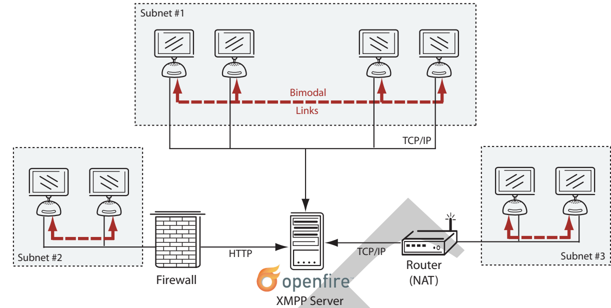
5.2. Implementation

A *Bimodal Link* is an ordinary XMPP private chat extended by the ability to transparently switch between ordinary indirect server-backed and direct out-of-band peer-to-peer communication over a Jingle-negotiated data channel. When the link is in *In-Band-Mode* mode, messages are delivered as part of the XMPP stream over the XMPP server. At configurable intervals the controller attempts to establish a P2P link using the Jingle facility provided as an extension to the SMACK library. If negotiation succeeds, the link switches to *Out-of-Band* mode, delivering messages directly without the indirection over the XMPP server. For that purpose each node runs a TCP server implemented on

DRAFT



(a) JXTA overlay network with rendezvous and relay infrastructure peers



(b) XMPP network with *Bimodal Links* and an XMPP server

Figure 2. JXTA / XMPP infrastructure comparison

top of MINA [4], that is responsible for handling incoming P2P connections. By leveraging MINA for connection management, we can benefit from it's high scalability and advanced QoS features (like TLS support and traffic shaping).

Manager	Space Complexity Client	Server	Time Complexity Update	Message Complexity Update
Complete	$O(\|G\|)$	$O(\|G\|)$	$O(\|G\|)$	$O(\|G\|)$
RBT	$O(v)$	$O(v\,\|G\|)$	$O(v \log(\|G\|))$	$O(v)$
Blind	$O(1)$	$O(\|G\|)$	$O(1)$	$O(1)$

Table 1. View type complexity characteristics.

6. Scalable Group Membership Management

Using the standard OPENFIRE MUCs as a group abstraction a lot of time and bandwidth is wasted on transmitting and processing status information, as a quadratic number of messages has to be send to inform each group member of each others membership status (e.g. join and leave operations). Additionally, the client keeps an account of all other group members, which leads to a memory usage linear in matters of the group size on each client. Consequently, the standard MUC implementation does not scale to thousands of hosts, neither client- nor server-side (see Section 8.1).

We enhance scalability of group management by limiting the *membership view* [13] of group members, i.e. the subset of group members about whose status a node is informed. Note that nodes can still unicast messages to all other nodes within the group, even if they are not in the local membership list, as long as they know their JID. We share the concept of partial membership lists with many DHT approaches (including Chord, Pastry and CAN), which use *neighbor sets* to route messages in a multi-hop fashion. While this approach scales well, multi-hop routing considerably increases communication latency, which is of prime importance in high performance computing. Messages in our architecture are routed in a single hop for directly connected nodes, in two hops for nodes connected to the same XMPP server and in a maximum of three hops, i.e. Node #1 → Server #1 → Server #2 → Node #2, for nodes connected to different XMPP servers.

6.1. Implementation

We developed a generic MUC, which delegates view management to separate *view managers*. We provide three manager implementations to satisfy the requirements for a broad range of use cases (custom managers can be integrated easily). Their complexity characteristics are summarized in Table 1.

Complete. This manager emulates the semantics of the standard XMPP MUC maintaining complete membership lists on all clients.

Red-Black Tree (RBT). This manager organizes group members in a red-black tree and notifies nodes only about the group presence of its adjoining nodes in the tree. Hence, if a node is not alone in a group, it will see a minimum of 1 and a maximum of 3 other members – we say its *view size* is 3. Each member can also configure its view size to other values, say v, which will lead to him being inserted $\min(1, \lfloor v/3 \rfloor)$ times with random keys into the red-black tree and thus having a maximum of v visible neighbors. Configurable view size allows for reflecting a nodes capabilities, i.e. available resources, or special responsibilities within a group, e.g. coordinators of distributed algorithms may need more detailed information about the groups composition. Updates in the tree (a joining or leaving member) can be computed in $O(v \log(|G|))$ time, where v is the maximum view size.

The properties of red-black trees allow for providing alternative implementations of our communication primitives

DRAFT

where the server is not involved: As the red-black tree is a spanning tree, we can easily realize a groupcast by propagating messages along its edges. Additionally, as a red-black tree is a binary-search tree, we can use host-to-host message routing for unicasts known from DHTs. The choice of using a red-black tree assures, that two members are always connected by at most $4v \log (|G|)$ other members, where $|G|$ is the size of group G. Thus, the path length is $O(v \log (|G|))$.

Blind. This manager does not send any information about other group members, which leads to optimal time and space usage, but minimum information. Note, that the group still enforces security restrictions and thus is appropriate to drive the root group realized in most communication frameworks with a hierarchical group model, e.g. the *world peergroup* in JXTA or the *universal group* in COHESION [22].

7. Efficient Communication

An XMPP server spends most time with XMPP stanza processing. Thus, an attempt to increase the overall performance of the communication subsystem should primarily address optimizing XML processing. In this section, we describe how we optimized the XML processing stack of OPENFIRE/SMACK by incorporating a binary XML encoding called *Fast Infoset* (FI) [2] to yield substantial latency and throughput improvements.

7.1. Fast Infoset

A major drawback of XML is that it is quite verbose. Since document size affects all stages in the XML processing chain (i.e. serialization, transmission, and parsing), techniques to reduce document size promise to increase processing performance. However, there is a trade-off between document size and pre-/postprocessing effort. Simple stream compression methods (e.g. GZIP) significantly reduce document size, but at the same time cause considerable pre-/postprocessing overhead. *Fast Infoset* overcomes this limitation by interweaving serialization and compression or decompression and parsing, respectively.

FI specifies a binary encoding format for the XML Information Set. It aims to provide more efficient serialization and parsing than the character-based standard XML format. FI is used to optimize both document size ($\approx 50\%$ on average compared to standard XML 1.0 serialization using Xerces 2.7.1 [2]) and processing performance ($\approx 25\%$ faster serialization and between 5 and 8 times faster parsing compared to Xerces 2.7.1 SAX) and thus is more advanced than simple stream compression based on GZIP used in contemporary XMPP servers. These improvements are achieved through exploiting redundancy by avoiding end-tags, applying string indexing and huffman encoding, aligning information for faster access and by directly embedding binary data into the stream, bypassing the usually necessary conversion to Base-64 representation.

7.2. Implementation

OPENFIRE leverages the *Reactor* design pattern [20] to achieve high scalability. Pending I/O-operations are handled sequentially by a single dispatcher thread (per CPU core). This processing model is very efficient, since fewer threads means less resource consumption and fewer context switches. However, since XMPP stanzas are delivered as elements embedded in a single large XML stream, this non-contiguous I/O processing style is problematic, as to our best knowledge, there are no non-blocking Java™XML parsers available. A non-blocking XML parser would return immediately when no more data is available from the underlying input stream, leaving the parser in a continuable state and allowing the dispatcher thread to continue processing pending operations from other connections.

To solve this problem, we dissect the incoming byte stream containing the XMPP stanzas. For that purpose we modified SMACK to prepend a header to each stanza specifying the length of the packet. Server-side logic reads the header and waits until the specified number of bytes have been received. As soon as the whole sequence of bytes has been received the whole stanza is forwarded at once to the XML parser, which immediately returns after the end tag concluding the XMPP stanza has been read.

8. Performance Evaluation

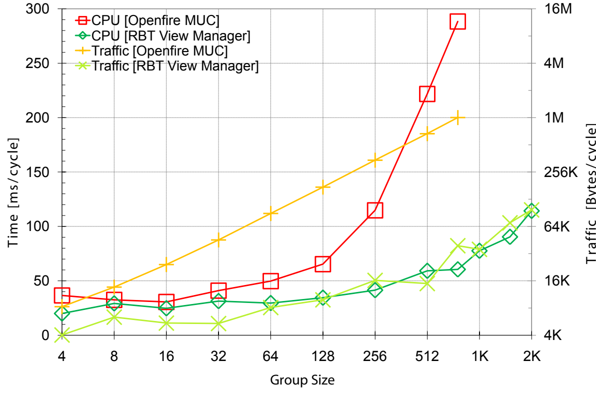
This section presents an analysis of the results obtained from running performance tests for both the original and our optimized SMACK/OPENFIRE XMPP stack.

The testbed consists of an AMD Athlon™ 64 X2 Dual Core 5600+ processor with 4 GB of RAM running the OPENFIRE XMPP server, and machines with Intel Pentium™ IV D processors (3.4 GHz) with 2 GB of RAM hosting the clients. All machines are running a 2.6 version Linux kernel and are connected by a Fast Ethernet local-area department network. A warm-up phase preceded each test run to eliminate the impact of *Just-In-Time* (JIT) compilation.

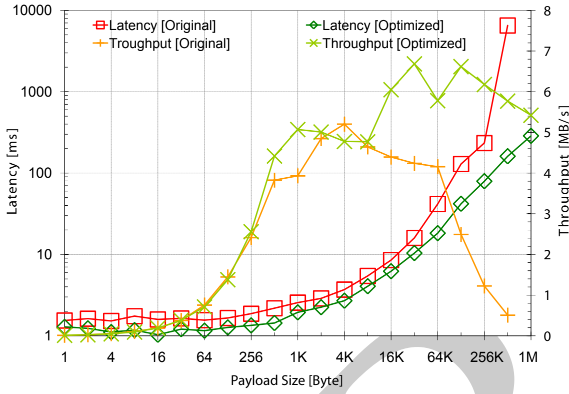
8.1. Membership Management

In this section, we report on the performance of the membership management operations, i.e. join and leave, described in Section 4. The test is carried out by having one of the group members join and leave periodically. To assess the suitability of our XMPP MUC room replacement for

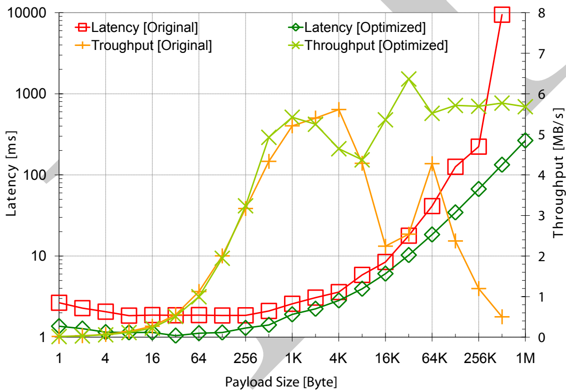
DRAFT



(a) Server-side resource usage per join/leave-cycle



(b) Unicast latency and throughput



(c) Groupcast latency and throughput

Figure 3. Performance measurement results

driving large-scale groups, we compare the server-side resource consumption, i.e. network traffic and CPU usage, of both the original view-complete and our RBT-based MUC (RBT-MUC) implementation with partial membership lists and view size 3. In order to obtain values related to a single join/leave-cycle, we calculate the quotient of overall resource consumption within the test period and the number of join/leave cycles actually performed. Figure 3a shows the results of our experiments. The theoretical linear message complexity of the original view-complete MUC room implementation is perfectly confirmed by our measurements. Both CPU time and network traffic increase linearly reaching a maximum of 288 ms/cycle and 1025 KB/cycle respectively for 768 nodes. Tests with node counts beyond 768 nodes failed due to excessive memory usage. The corresponding values of the RBT-MUC implementation are 77 ms/cycle and 39 KB/cycle respectively, resulting in improvements (growing with group size) between 82% and 376%. From these results, we conclude that view completeness becomes expensive for groups growing larger than $\approx$ 128 nodes. The numbers substantiate that the membership management operations for our RBT view manager are efficient enough to support groups with thousands of occupants. Note, that a recent study on Desktop Grids [16] shows that the mean host session time ranges between 2.8 hours for weekdays and 5.9 hours for weekends. Given these numbers, the CPU utilization for a 2K (110 ms/cycle) group managed by our RBT view manager would approximately range between 0.5% and 1.1% on our dual-core server machine. These results substantiate that our solution supports a significant number of concurrent groups of considerable size for real-world churn rates.

8.2. Communication

In this section, we report on the results of latency and bandwidth tests for the basic communication primitives. Our tests only address relayed communication, since out-of-band delivery of messages is based on plain TCP/IP.

The bandwidth test is carried out by having the sender node emitting a large number of messages to a single receiver node (unicast) or to all the members of the group (groupcast). Either the single receiver for unicasts or the last receiver for groupcasts acknowledges the receipt of all messages after the last message has arrived. Throughput is calculated as the quotient between transmitted payload bytes and the time elapsed between emission of the first message and receipt of the acknowledgment. Obviously OPENFIRE/SMACK has problems in dealing with large messages (see Figures 3b and 3c). Due to a hardcoded server side limit on message size to prevent denial of service attacks, sending messages larger than 512K is impossible. For large messages between 16 KB and 512 KB our optimizations result in an improvement between 38% and 1038%

DRAFT

for unicasts and between 130% and 1051% for groupcasts. With a resulting throughput of approximately 6.7 MB/s for unicast and 6.4 MB/s for groupcasts our substrate outperforms the reference implementation with 5.2 MB/s for unicasts and 5.6 MB/s for groupcasts. While our implementation is roughly on par with the reference implementation for small messages (i.e. [1 B, 256 B]), we achieve up to 29% improved throughput rates for unicasts and up to 13% for groupcasts when mid-size messages (i.e. [512 B, 8 KB]) are transmitted. This indicates that the dissection of the input stream as described in Section 7 adds no significant overhead to the parsing process.

Latency measurements are carried out in a ping-pong fashion. The sender emits a message that is echoed back by the recipient. For groupcasts only the last recipient responds. The one-way latency is calculated as the time between emission of the message and receipt of the acknowledgment divided by two. Note that the latency numbers given here are actually two hop latencies (i.e. sender $\rightarrow$ server $\rightarrow$ receiver). Our optimizations result in very pronounced improvements for both unicasts and groupcasts and all messages sizes. Improvements in unicast latency are between 16% and 35% for small- and mid-size and 27% and 97% for large messages. The increases for groupcasts are between 21% and 48% for small- and mid-size and range from 28% to 99% for large messages. With 1.28 ms (unicast) and 1.36 ms (groupcasts), the minimal achievable latencies (i.e. for a message carrying no payload) are noticeably lower for our optimized implementation than for the reference implementation (1.53 ms and 2.65 ms respectively). The unicast latency is comparable to the latency of JXTA sockets [8] in the case of direct communication.

9. Related Work

OurGrid [12] is a platform for P2P Grid Computing. Currently (version 3.3), OurGrid uses *Remote Method Invocation* (RMI) over TCP/IP as programming model. RMI suffers from poor performance, due to a large protocol overhead and also does not integrate well with restrictive firewalls and NAT devices. In the upcoming 4.0 release, OurGrid will switch to an asynchronous programming model called JDIC [17] that employs XMPP as transport protocol.

The *Distributed Infrastructure with Remote Agent Control* (Dirac) [25] is a Service Oriented Architecture (SOA) composed of lightweight services forming a scalable robust Grid Computing environment to manage and track a large number of computing tasks. It differs from applications our substrate is primarily targeted for by being tailored for high-throughput instead of high-performance computing. XMPP is used to implement three different aspects of the system: inter-service messaging, state monitoring of agents and job-level monitoring. While plain XMPP is suitable for the first two applications as the number of services (5-20) and agents (10-100) is low, the third is more critical as thousands of jobs are active at peak time. In contrast to our approach of improving and extending the XMPP protocols, Dirac restricts itself to protect the system from the impact of overloaded XMPP servers by applying virtualization techniques. Certainly, Dirac could benefit from adopting our improved network substrate.

JXTA [24] is an open source initiative, sparked and maintained by Sun Microsystems™. It's primary goal is to provide a foundation for interoperable P2P applications. JXTA consists of a set of six language- and platform-independent protocol specifications. It provides basic services for generic P2P applications including peer group organization, inter-peer communication, and resource discovery. Note, that JXTA is neither limited to nor optimized for P2P Grid Computing. While it provides higher-level services like discovery and monitoring, it lacks explicit support for membership views, which we consider fundamental for high performance computing. However, both JXTA and XMPP provide similar functionality on the lower protocol layers dedicated to communication. Although JXTA is the most advanced P2P library currently available and is considered to be a good choice for implementing distributed computing platforms [8], performance studies have revealed weaknesses in the Java™ implementation of the version 2.0 protocol specification. That includes pipe latency and throughput for small messages [14], reliability of TCP connections [23], and rendezvous network stability [11].

Peer-to-Peer Simplified (P2PS) [26] is an open-source project providing an infrastructure for P2P service discovery and pipe-based communication. The P2PS reference implementation is written in Java™. While sharing many concepts with JXTA, P2PS is more focused on simplicity than on feature richness. P2PS peers can communicate over multiple protocols that can be replaced transparently to the application. P2PS service discovery is based on XML advertisements and queries. P2PS discovery uses subnets to broadcast advertisements and queries efficiently. As in JXTA, rendezvous peers are responsible for caching and forwarding advertisements and queries to rendezvous' in other subnets. Although P2PS provides a peer group abstraction, there is no support for broadcasting within groups. Thus, a substantial requirement for many distributed algorithms is not satisfied. To our knowledge there is no performance evaluation available for P2PS.

10. Conclusion

Supporting non-trivial parallelism for Desktop Grid Computing requires efficient P2P interaction among the nodes. This characteristic feature must ultimately be enabled by the substrate technology. Such a substrate must

DRAFT

satisfy a set of functional (i.e. membership management and fundamental communication primitives) and non-functional requirements (i.e. performance, scalability, connectivity, and security) to be able to serve as a basic building block for existing and future P2P Grid Computing platforms.

In analogy to the development of Grid Computing platforms, which experienced a phase of consolidation through standardization in the last decade, we believe that P2P Grid Computing has to pass through a similar process by adopting existing open standards to tap its full potential. In this paper, we demonstrate that a substrate based on the XMPP standard can fulfill all identified functional and non-functional requirements.

References

[1] ejabberd - The Erlang Jabber/XMPP daemon community site. http://www.ejabberd.im (04/24/2008).

[2] Fast Infoset Project. http://fi.dev.java.net (04/24/2008).

[3] Ignite Realtime - A Jive Software Community. http://www.igniterealtime.org (04/24/2008).

[4] MINA. http://mina.apache.org (04/24/2008).

[5] The XMPP Software Foundation. http://www.xmpp.org (04/24/2008).

[6] D. P. Anderson. BOINC: A System for Public-Resource Computing and Storage. In *5th IEEE/ACM International Workshop on Grid Computing*, Pittsburgh, USA, 2004.

[7] D. P. Anderson and G. Fedak. The computational and storage potential of volunteer computing. In *Proc. of the Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2006)*, pages 73–80, Singapore, 2006.

[8] G. Antoniu, P. Hatcher, M. Jan, and D. A. Noblet. Performance evaluation of jxta communication layers. In *Proceedings of the Fifth IEEE International Symposium on Cluster Computing and the Grid (CCGrid'05)*, pages 251–258, Washington, DC, USA, 2005. IEEE Computer Society.

[9] W. Blochinger, C. Dangelmayr, and S. Schulz. Aspect-oriented parallel discrete optimization on the cohesion desktop grid platform. In *Proc. of the Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2006)*, pages 49–56, Singapore, May 2006.

[10] W. Blochinger, W. Westje, W. Küchlin, and S. Wedeniwski. ZetaSAT – Boolean satisfiability solving on desktop grids. In *Proc. of the Fifth IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2005)*, volume 2, pages 1079–1086, Cardiff, UK, May 2005.

[11] K. Burbeck, D. Garpe, and S. Nadjm-Tehrani. Scale-up and performance studies of three agent platforms. In *IEEE International Conference on Performance, Computing, and Communications*, pages 857– 863, 2004.

[12] W. Cirne, F. Brasileiro, N. Andrade, L. Costa, A. Andrade, R. Novaes, and M. Mowbray. Labs of the World, Unite!!! *Journal of Grid Computing*, 4(3):225–246, 2006.

[13] A. J. Ganesh, A.-M. Kermarrec, and L. Massoulié. Peer-to-Peer Membership Management for Gossip-based Protocols. *IEEE Trans. Comput.*, 52(2):139–149, 2003.

[14] E. Halepovic and R. Deters. The costs of using jxta. In *Proceedings of the 3rd International Conference on Peer-to-Peer Computing*, page 160, Washington, DC, USA, 2003. IEEE Computer Society.

[15] D. Kondo, G. Fedak, F. Cappello, A. A. Chien, and H. Casanova. On resource volatility in enterprise desktop grids. In *Proceedings of the Second IEEE International Conference on e-Science and Grid Computing*, page 78, Washington, DC, USA, 2006. IEEE Computer Society.

[16] D. Kondo, M. Taufer, C. L. B. III, H. Casanova, and A. A. Chien. Characterizing and evaluating desktop grids: An empirical study. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS)*, volume 01, page 26b, Los Alamitos, CA, USA, 2004. IEEE Computer Society.

[17] A. Lima, W. Cirne, F. V. Brasileiro, and D. Fireman. A Case for Event-Driven Distributed Objects. In *OTM Conferences (2)*, pages 1705–1721, 2006.

[18] M. Ripeanu, A. Iamnitchi, and I. Foster. Mapping the gnutella network. *IEEE Internet Computing*, 06(1):50–57, 2002.

[19] P. Saint-Andre. End-to-End Signing and Object Encryption for the Extensible Messaging and Presence Protocol (XMPP). RFC 3923 (Proposed Standard), Oct. 2004.

[20] D. C. Schmidt. Reactor: an object behavioral pattern for concurrent event demultiplexing and event handler dispatching. pages 529–545, 1995.

[21] S. Schulz and W. Blochinger. An integrated approach for managing peer-to-peer desktop grid systems. In *Proc. of the Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2007)*, pages 233–240, Rio de Janeiro, Brazil, May 2007.

[22] S. Schulz, W. Blochinger, M. Held, and C. Dangelmayr. COHESION - A microkernel based desktop grid platform for irregular task-parallel applications. *Future Generation Computer Systems – The International Journal of Grid Computing: Theory, Methods and Applications*, 2008.

[23] J.-M. Seigneur. JXTA Pipe Performance. http://bench.jxta.org (04/24/2008).

[24] B. Traversat, A. Arora, M. Abdelaziz, M. Duigou, C. Haywood, J.-C. Hugly, E. Pouyoul, and B. Yeager. Project JXTA 2.0 Super-Peer Virtual Network. Technical report, Sun Microsystems, May 2003.

[25] A. Tsaregorodtsev, V. Garonne, and I. Stokes-Rees. Dirac: A scalable lightweight architecture for high throughput computing. In *GRID '04: Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing*, pages 19–25, Washington, DC, USA, 2004. IEEE Computer Society.

[26] I. Wang. P2PS (Peer-to-Peer Simplified). In *Proceedings of 13th Annual Mardi Gras Conference - Frontiers of Grid Applications and Technologies*, pages 54–59. Louisiana State University, February 2005.

DRAFT