Project no. IST-033576

XtreemOS

Integrated Project
BUILDING AND PROMOTING A LINUX-BASED OPERATING SYSTEM TO SUPPORT VIRTUAL ORGANIZATIONS FOR NEXT GENERATION GRIDS

Reproducible Evaluation of the publish subscribe system

D3.2.7

Due date of deliverable: November 30^{th}, 2008
Actual submission date: November 13^{th}, 2008

Start date of project: June 1^{st} 2006

Type: Deliverable
WP number: WP3.2
Task number: T3.2.2

Responsible institution: ZIB
Editor & and editor's address: Thorsten Schütt
Zuse Institute Berlin
Takustrasse 7
14195 Berlin
Germany

Version 0.5 / Last edited by Thorsten Schütt / Nov 24^{th}, 2008

Project co-funded by the European Commission within the Sixth Framework Programme		
Dissemination Level		
PU	Public	√
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

Revision history:

Version	Date	Authors	Institution	Section affected, comments
0.1	2008/10/06	Thorsten Schütt	ZIB	first draft
0.2	2008/10/23	Thorsten Schütt	ZIB	added benchmark results
0.3	2008/10/24	Thorsten Schütt	ZIB	clean-up
0.4	2008/11/11	Thorsten Schütt	ZIB	applied reviewer suggestions
0.5	2008/11/13	Thorsten Schütt	ZIB	applied reviewer suggestions

Reviewers:
Philip Robinson (SAP),Jonathan Marti (BSC)

Tasks related to this deliverable:

Task No.	Task description	Partners involved°
T3.2.2	Design and implementation of a scalable publish/subscribe system	ZIB*

°This task list may not be equivalent to the list of partners contributing as authors to the deliverable
*Task leader

Executive summary

This deliverable presents the state of the design and evaluation of the XtreemOS software component "Publish/Subscribe System". This system is a key component of the highly available and scalable infrastructure as described in deliverable D3.2.1 under the responsibility of WP 3.2.

Scalaris is made up of four layers which together implement the "Publish/Subscribe System". At the bottom is a distributed hash table (DHT) which provides a simple put and get interface to a dictionary like data-structure which is distributed over all participating nodes. The DHT provides scalability and fault-tolerance.

The second layer implements so called symmetric replication which guarantees the availability of data even when nodes fail or are unavailable. Symmetric replication divides all nodes into r equivalence classes, and distributes the replica so that the nodes storing the replicas of item belong to different equivalence classes.

On top of the replication layer, we implemented a transaction data access layer, which performs all read and write operations inside of transactions. The transactions allow us to consistently update all replicas belonging to one item and at the same update several items in one atomic operation. The transaction framework employs Paxos.

The final layer is the "Publish/Subscribe System", which uses the layers below for managing subscribers and topics. In Sec. 3, we run several tests with different numbers of nodes for evaluating the scalability. We will show that the Pub/Sub system scales linearly with the number of nodes.

Contents

1 Introduction

One important goal of Workpackage 3.2 of the XtreemOS project is to provide a *highly scalable publish/subscribe service* (pub/sub). This service will be used by XtreemOS services to notify other services and users about important, possibly time-critical events within the XtreemOS operating system. A typical event could be an unexpected termination of a job, an update in the file system, or the availability of new resources and services. One potential domain where XtreemOS can be applied is in cloud computing and the provision of Web 2.0 services. XtreemOS therefore needs to be capable of supporting services that have high numbers and frequencies of transactional requests and notifications. Hence, the herewith described pub/sub system is at the core of many services in XtreemOS.

We will give a general overview of our "Publish/Subscribe System" called Scalaris in Sec. 2. Scalaris is based on a distributed hashtable and transactional data access layer on top. In Sec. 3 we will show that Scalaris scales linearly with the number of nodes.

2 System Description

Web 2.0, that is, the Internet as an information society platform supporting business, recreation and knowledge exchange, initiated a business revolution. Service providers offer Internet services for shopping (Amazon, eBay), online banking, information (Google, Flickr, Wikipedia), social networking (MySpace, Facebook), and recreation (Second Life, online games). In our information society, Web 2.0 services are no longer just nice to have, but customers depend on their continuous availability, regardless of time and space.

How to cope with such strong demands, especially in case of interactive community services that cannot be simply replicated? All users access the same Wikipedia, meet in the same Second Life environment and want to discuss with others via Twitter. Even the shortest interruption, caused by system downtime or network partitioning may cause huge losses in reputation and revenue. Web 2.0 services are not just an added value, but they must be dependable. Apart from 24/7 availability, providers face another challenge: they must, for a good user experience, be able to respond within milliseconds to incoming requests, regardless whether thousands or millions of concurrent requests are currently being served. Indeed, scalability is a key challenge. Any scalable service, to be affordable, somehow requires the system to be self managing.

Our Scalaris system, described below, provides a comprehensive solution for self managing, scalable data management and pub/sub systems. We expect Scalaris and similar systems to become an important core service of future Cloud Computing environments.

As a common key aspect, all Web 2.0 services have to deal with concurrent data updates. Typical examples are checking the availability of products and their prices, purchasing items and putting them into virtual shopping carts, and updating the state in multi-player online games. Clearly, many of these data operations have to be atomic, consistent, isolated and durable (so-called ACID properties). Traditional centralized database systems are ill-suited for this task, sooner or later they become a bottleneck for business workflow. Rather, a scalable, transactional data store like Scalaris is what is needed.

2.1 Scalaris

We set out to build a distributed key/value store capable of serving thousands or even millions of concurrent data accesses per second. Providing strong data consistency in the face of node crashes and hefty concurrent read and write accesses was one of our major goals.

With our Scalaris system, we do not attempt to replace current database management systems with their general, full-fledged SQL interfaces. Instead our tar-

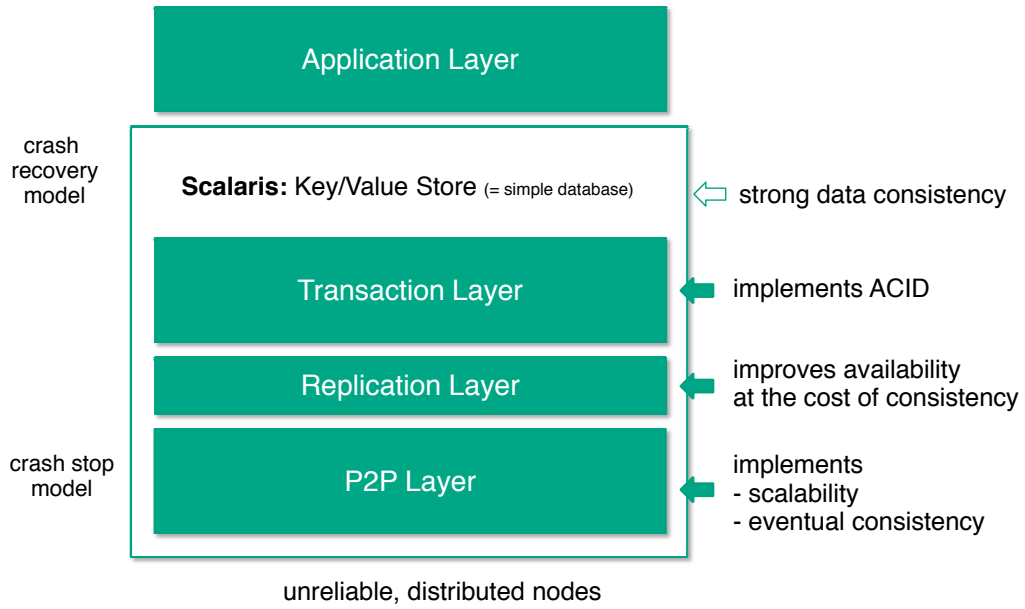

Figure 1: Scalaris system architecture.

get is to support transactional Web 2.0 services like those needed for Internet shopping, banking, or multi-player online games. Our system consists of three layers:

- At the bottom, an enhanced structured overlay network, with logarithmic routing performance, provides the basis for storing and retrieving keys and their corresponding values. In contrast to many other overlays, our implementation stores the keys in lexicographical order. Lexicographical ordering instead of random hashing enables control of data placement which is necessary for low latency access in multi-datacenter environments.
- The middle layer implements data replication. It enhances the availability of data even under harsh conditions such as node crashes and physical network failures.
- The top layer provides transactional support for strong data consistency in the face of concurrent data operations. It uses a fast consensus protocol with low communication overhead that has been optimally embedded into the structured overlay.

As illustrated in Fig. 1, these three layers provide a distributed key/value store as a scalable and highly available service which is an important building block for

Web 2.0 applications. One of the applications, we are running on Scalaris is the pub/sub service for XtreemOS. The following sections describe the layers in more detail.

2.1.1 P2P Overlay

At the bottom layer, we use the structured overlay protocol Chord$^{\#}$ [15, 16] for storing and retrieving key-value pairs in nodes (peers) that are arranged in a virtual ring. In each of the N nodes, Chord$^{\#}$ maintains a routing table with $O(\log N)$ entries (fingers). In contrast to Chord [20], Chord$^{\#}$ stores the keys in lexicographical order, thereby allowing range queries. To ensure logarithmic routing performance, the fingers in the routing table are computed in such a way that successive fingers in the routing table cross an exponentially increasing number of nodes in the ring.

Chord$^{\#}$ uses the following algorithm for computing the fingers in the routing table (the infix operator $x\ .\ y$ retrieves y from the routing table of a node x):

$$\mathit{finger}_i = \begin{cases} \mathit{successor} & : i = 0 \\ \mathit{finger}_{i-1}\ .\ \mathit{finger}_{i-1} & : i \neq 0 \end{cases}$$

Thus, to calculate the i^{th} finger, a node asks the remote node listed in its $(i-1)^{th}$ finger to which node his $(i-1)^{th}$ finger refers to. In general, the fingers in level i are set to the fingers' neighbors in the next lower level $i-1$. At the lowest level, the fingers point to the direct successors. The resulting structure is similar to a skip list, but the fingers are computed deterministically without any probabilistic component.

Compared to Chord, Chord$^{\#}$ does the routing in the *node space* rather than the *key space*. This finger placement has two advantages over that of Chord: First, it works with any type of keys as long as a total order over the keys is defined, and second, finger updates are cheaper, because they require just one hop instead of a full search (as in Chord). A proof of Chord$^{\#}$'s logarithmic routing performance can be found in [15].

2.1.2 Replication and Transaction Layer

The scheme described so far provides scalable access to distributed key/value pairs. To additionally tolerate node failures, we replicate all key/value pairs over r nodes using symmetric replication [5]. Read and write operations are performed on a majority of the replicas, thereby tolerating the unavailability of up to $\lfloor (r-1)/2 \rfloor$ nodes.

Each item is assigned a version number. Read operations select the item with the highest version number from a majority of the replicas. Thus a single read operation accesses $\lceil (r+1)/2 \rceil$ nodes, which is done in parallel.

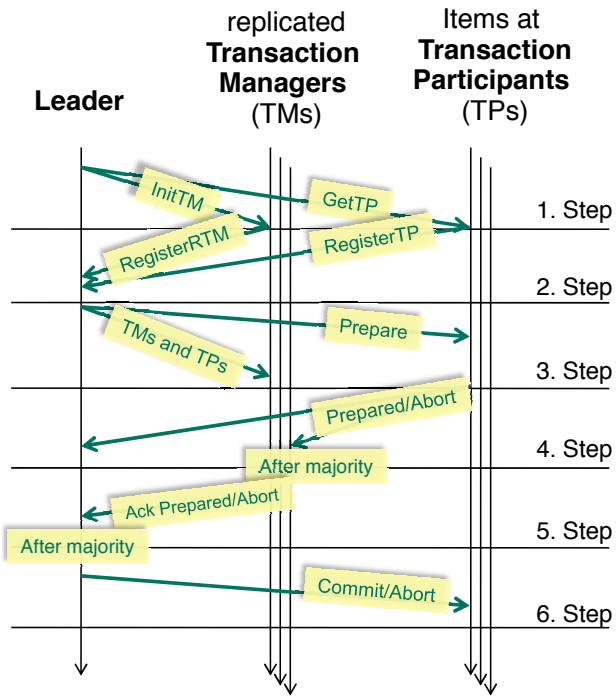

Figure 2: Adapted Paxos used in Scalaris.

Write operations are done with an adapted Paxos atomic commit protocol [12]. In contrast to the 3-Phase-Commit protocol (3PC) used in distributed database systems, the adapted Paxos is non-blocking, because it employs a group of *acceptors* rather than a single transaction manager. We select those nodes as acceptors that are responsible for symmetric replication of the transaction manager. The group of acceptors is determined by the transaction manager just before the prepare request is sent to the transaction participants (Fig. 2). This gives a pseudo static group of transaction participants at validation time, which is contacted in parallel.

Write operations and transactions need three phases, including the phase to determine the nodes that participate in the atomic commit. For details see [12, 19].

In Scalaris, the adapted Paxos protocol serves two purposes: First it ensures that all replicas of a *single* key are updated consistently, and second it is used for implementing transactions over *multiple* keys, thereby realizing the ACID properties (atomicity, concurrency, isolation, durability).

2.2 Self-Management

For many Web 2.0 services, the total cost-of-ownership is dominated by the costs needed for personnel to maintain and optimize the service. Scalaris greatly reduces the operation cost with its built-in self* properties:

- *Self healing:* Scalaris continuously monitors the hosts it is running on. When it detects a node crash, it immediately repairs the overlay network and the database. Management tasks such as adding or removing hosts require minimal human intervention.
- *Self tuning:* Scalaris monitors the nodes' workload and autonomously moves items to distribute the load evenly over the system to improve the response time of the system. When deploying Scalaris over multiple data-centers, these algorithms are used to place frequently accessed items nearby the users.

This protection scheme helps in high stress situations but it also constantly monitors the system and proactively repairs and tunes the system before larger interruptions can occur. In traditional database systems these operations require human interference which is error prone and costly. With Scalaris the same number of system administrators can operate much larger installations than with legacy databases.

2.3 Implementation

Because of asynchronous communication and unreliable networks, distributed algorithms are difficult to implement and the resulting code is error-prone. Using imperative programming languages and message passing libraries it is easy to implement deadlocks of lifelocks.

In the literature [6], the *actor model* [7] became a popular paradigm for describing and reasoning about distributed algorithms. Chord$^{\#}$ and the transaction algorithms in Scalaris were also developed according to this model. The basic primitives in this model are actors and messages. Every actor has a state, can send messages, act upon messages and spawn new actors.

These primitives can be easily mapped to Erlang processes and messages. The close relationship between the theoretical model and the programming language allows a smooth transition from the theoretical model to prototypes and eventually to a complete system.

Our Erlang implementation of Scalaris comprises many components. It has a total of 11,000 lines of code: 7,000 for the P2P layer with replication and basic system infrastructure, and 2,700 lines for the transaction layer.

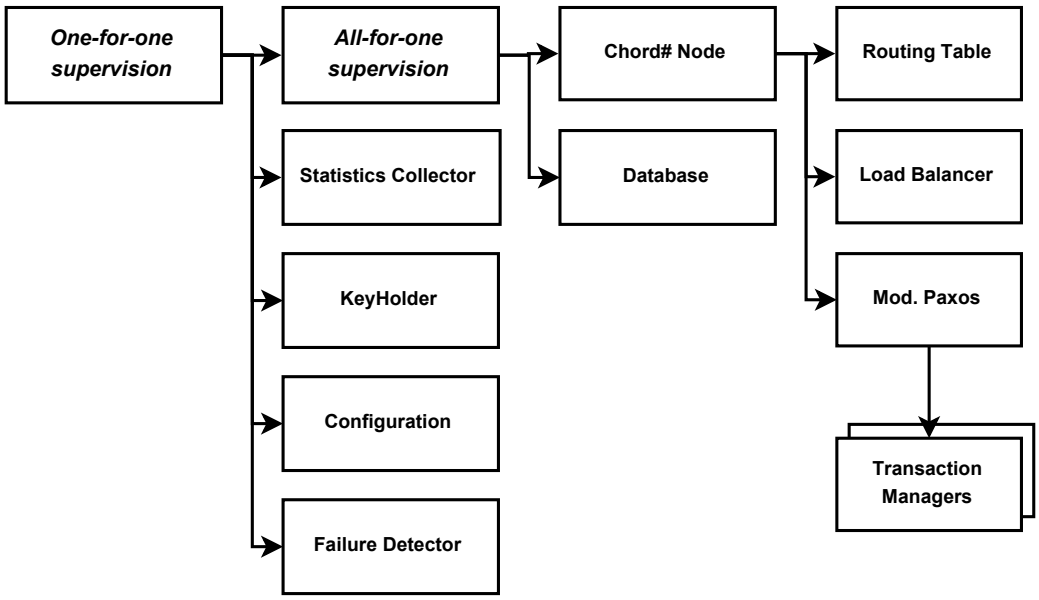

Figure 3: Supervisor tree of a Scalaris node. Each box represents one process.

2.3.1 Components and Supervisor Tree

Scalaris is a distributed algorithm. Each peer runs a number of processes as shown in Fig. 3:

- *Failure Detector* supervises other peers and sends a crash message when a node failure is detected.
- *Configuration* provides access to the configuration file and maintains parameter changes made at runtime.
- *Key Holder* stores the identifier of the node in the overlay.
- *Statistics Collector* collects statistics and forwards them to central statistic servers.
- Chord$^{\#}$ Node performs all important functions of the node. It maintains, among other things, the successor list and the routing table.
- *Database stores* the key-value pairs of this node. The current implementation uses an in-memory dictionary, but disk store based on DETS or Mnesia could also be used.

The processes are organized in a supervisor tree as illustrated in Fig. 3. The first four processes are supervised by a *one-for-one supervisor* [1]: When a slave crashes, it is restarted by the supervisor. The right-most processes (Chord$^{\#}$ Node and Database) are supervised by an *all-for-one supervisor* which restarts *all* slaves when a single slave crashed. In Scalaris, when either of the Chord$^{\#}$ Node or the Database process fails, the other is explicitly killed and both are restarted to ensure consistency.

2.4 Wikipedia on Scalaris

As a challenging benchmark for Scalaris, we implemented the core of Wikipedia, the "free encyclopedia, that anyone can edit". Wikipedia runs on three sites. The main one in Tampa is organized in three layers, the proxy server layer, the web server layer, and the MySQL database layer. The proxy layer serves as a cache for recent requests, and the web server layer runs the application logic and issues requests to the data base layer. Wikipedia handles about 50,000 requests per second, from which 48,000 are cache hits in the proxy server layer and 2,000 are processed by the data base layer. The proxy and the web server layers are embarrassingly parallel and therefore trivial to scale. From a scalability point of view, only the data base layer is challenging.

Our implementation uses Scalaris to replace the data base layer. This enables us to run Wikipedia on geographically distributed sites and to scale to almost any number of hosts. It inherits all the favorable properties of Scalaris, such as scalability and self management.

The Wikipedia on Scalaris is fast. Using eight servers it executes 2,500 transactions per second. All operations are performed within transactions to guarantee data consistency and replica synchronization. Adding more computers improves the performance almost linearly. The public Wikipedia, in contrast, employs ten servers to execute 2,000 requests per second on the large master/slave MySQL database in Tampa.

3 Evaluation

Scalaris has a built-in benchmarking facility – the bench_server. The module provides functions for executing benchmarks on all nodes, on which Scalaris is currently running.

All benchmarks were run on an Intel Cluster at ZIB. Each node has 2 Quad-Core E5420s running at 2.5GHz and 16GB memory. The nodes are connected via GigE and Infiniband. All tests were performed on the GigE network.

If not noted otherwise, all tests were run with one Erlang virtual machine per cluster node and 16 Scalaris nodes per virtual machine. As there are 8 cores per node, we are running 2 Scalaris nodes per core.

For the tests we simulated two scenarios: a) publish and b) subscribe:

Publish The publish operation is one transaction which reads the list of a topic's subscribers.

Subscribe The subscribe operation is one transaction which reads the old list of a topic's subscribers topic, changes the list and writes it back.

Each item is stored in four replicas.

All benchmarks involved the following five steps:

1. Start stop watch
2. Start n threads in each VM
3. Each threads executes the operation to test i times
4. Wait for all threads to finish
5. Stop stop watch

The variable n is plotted on x-axis of the graphs as the number of threads per VM. Note, that the scales are semilogarithmic. For the publish operation 2000 iterations were executed while the subscribe operation was run 100 times.

All tests were run using Revision 56 of Scalaris(http://code.google.com/p/scalaris).

3.1 1 Node

The one node test is different from the remaining tests, as all communication is locally within the same virtual machine and no TCP/IP was involved. The reduced overhead results in higher throughput.

The publish operation is simple enough, that a single thread issuing the requests provides the best performance. The subscribe benefits from the additional

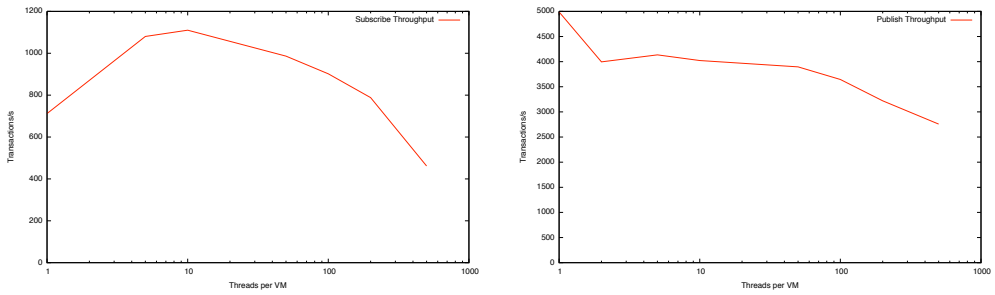

Figure 4: 1 node: a) Subscribe and b) Publish

parallelism of several concurrent requests. The subscribe operation needs more messages to execute and therefore takes more time to finish. The additional concurrency allows to hide the latency.

3.2 2 Nodes

In the tests with two nodes the subscribe operation is slightly slower than with only one node, because of the additional overhead of the network. Similar to the one node test a medium amount of parallelism in the request issuers provides the best throughput.

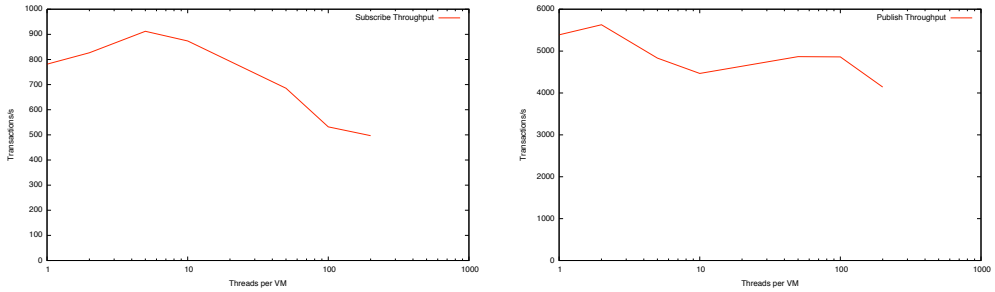

Figure 5: 2 nodes: a) Subscribe and b) Publish

The publish operation is slightly faster with two nodes and is pretty independent of the grade of parallelism.

3.3 4 Nodes

The subscribe performance again peaks at 10 threads per VM.

The publish performance is again pretty independent of the grade of parallelism.

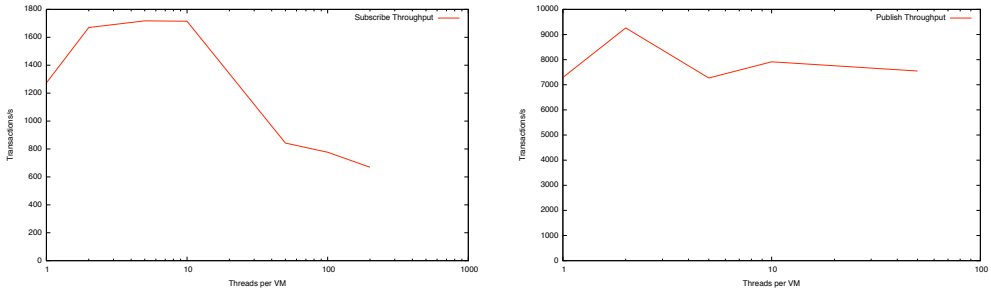

Figure 6: 4 nodes: a) Subscribe and b) Publish

3.4 8 Nodes

The subscribe performance continues to increase almost linearly with the number of nodes and can handle high levels of concurrency.

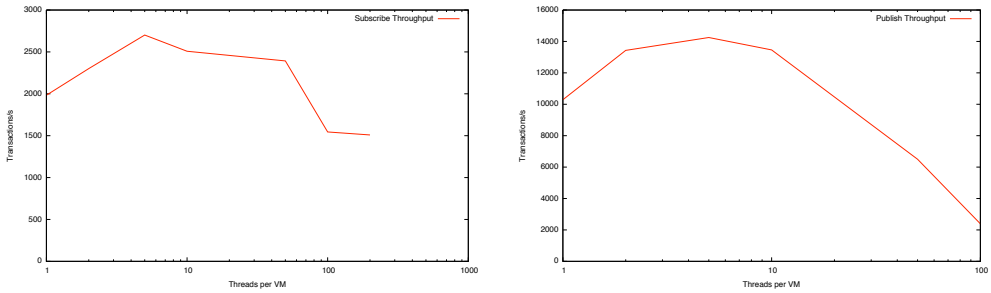

Figure 7: 8 nodes: a) Subscribe and b) Publish

The publish performance is steady up to 10 threads per VM, but drops for higher levels of concurrency. Note though, 100 threads per VM means that 8*100 threads issue publish requests as fast as they can.

3.5 16 Nodes

The subscribe performance peaks at 50 threads per VM or 800 concurrent users.

We stopped plotting the publish performance with high levels of concurrency because the execution times of several runs with the same parameters varied to much.

3.6 Scaling

Over a wide range of system sizes the system scales linearly. The exception is the single node scenario because the absents of network overhead.

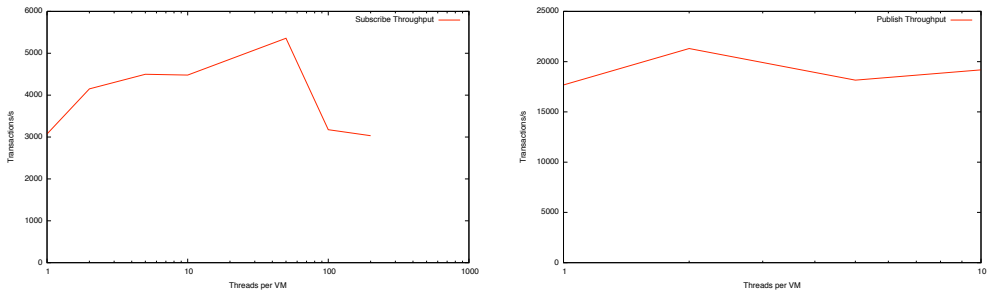

Figure 8: 16 nodes: a) Subscribe and b) Publish

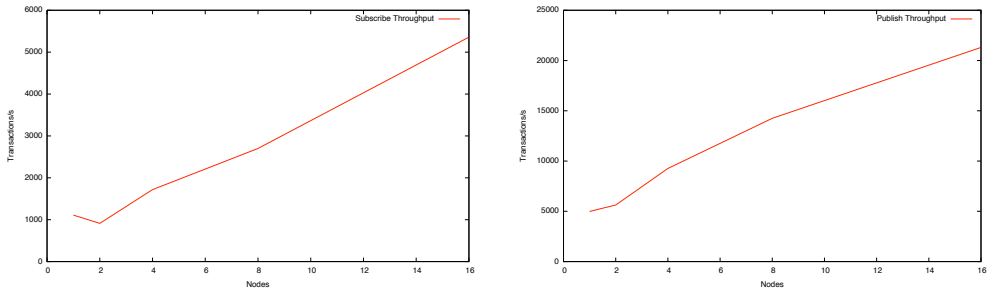

Figure 9: Scaling: a) Subscribe and b) Publish

4 Current Status and Deployment

In the last couple of month, we promoted Scalaris at several high-profile events:

21.05.2008 IEEE Scale Challenge
14.06.2008 Google Scalability Conference
27.06.2008 Erlang eXchange
23.07.2008 Open Source Release
27.09.2008 ACM Sigplan Erlang Workshop

Since July, Scalaris is available as open-source at: http://code.google.com/p/scalaris and we hope to build a community of users from XtreemOS partners as well as external groups. At the aforementioned conferences, we were contacted by several companies who expressed interest in using Scalaris.

5 Conclusion

Scalaris provides a scalable and self managing pub/sub system and a transactional key-value store. Its scalability and self* capabilities were demonstrated in the IEEE Scalable Computing Challenge 2008, where Scalaris won the 1st prize (Fig. 10).

Figure 10: Scalaris won the 1st price at the IEEE Scalable Computing Challenge 2008.

The pub/sub service will be included in the next release of XtreemOS and it will be first integrated with XtreemFS.

Compared to other data services, Scalaris has significantly lower operating costs. Scalaris and similar systems will be an important building block for future Cloud Computing environments and Grid services.

References

[1] J. Armstrong. *Programming Erlang: Software for a Concurrent World.* Pragmatic Programmers, ISBN: 978-1-9343560-0-5, July 2007

[2] R. Baldoni, L. Querzoni, A. Virgillito, R. Jiménez-Peris, and M. Patiño-Martínez. *Dynamic Quorums for DHT-based P2P Networks.* NCA, pp. 91–100, 2005.

[3] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's Highly Available Key-Value Store *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, Oct. 2007.

[4] JJ Furman, J. S. Karlsson, J. Leon, A. Lloyd, S. Newman, and P. Zeyliger. Megastore: A Scalable Data System for User Facing Applications. *SIGMOD 2008*, Jun. 2008.

[5] A. Ghodsi, L. O. Alima, and S. Haridi. Symmetric Replication for Structured Peer-to-Peer Systems. *3rd Intl. Workshop on Databases, Information Systems and P2P Computing*, 2005.

[6] R. Guerraoui and L. Rodrigues. Introduction to Reliable Distributed Programming. Springer-Verlag 2006.

[7] C. Hewitt, P. Bishop, and R. Steiger. A Universal Modular ACTOR Formalism for Artificial Intelligence. IJCAI, 1973.

[8] M. Hoegqvist, S. Haridi, N. Kruber, A. Reinefeld and T. Schütt. Using Global Information for Load Balancing in DHTs. *Workshop on Decentralized Self Management for Grids, P2P, and User Communities*, Oct. 2008.

[9] A. Lakshman, P. Malik, and K. Ranganathan. Cassandra: A Structured Storage System on a P2P Network. *SIGMOD 2008*, Jun. 2008.

[10] L. Lamport. Fast Paxos. *Distributed Computing* 19(2):79–103, 2006.

[11] M. M. Masud and I. Kiringa. *Maintaining consistency in a failure-prone P2P database network during transaction processing.* Proceedings of the 2008 International Workshop on Data management in peer-to-peer systems, pp. 27–34, 2008.

[12] M. Moser and S. Haridi. Atomic Commitment in Transactional DHTs. *1st CoreGRID Symposium*, Aug. 2007.

[13] S. Plantikow, A. Reinefeld, and F. Schintke. Transactions for Distributed Wikis on Structured Overlays. *18th IFIP/IEEE Distributed Systems: Operations and Management (DSOM 2007)*, Oct. 2007.

[14] Scalaris code: http://code.google.com/p/scalaris/.

[15] T. Schütt, F. Schintke, and A. Reinefeld. Structured Overlay without Consistent Hashing: Empirical Results. *GP2PC'06*, May 2006.

[16] T. Schütt, F. Schintke, and A. Reinefeld. A Structured Overlay for Multi-Dimensional Range Queries. *Europar*, Aug. 2007.

[17] T. Schütt, F. Schintke, and A. Reinefeld. Scalable Wikipedia with Erlang. *Google Scalability Conference*, Jun. 2008.

[18] T. Schütt, F. Schintke, and A. Reinefeld. Scalaris: Reliable Transactional P2P Key/Value Store. ACM SIGPLAN Erlang Workshop, Sep. 2008.

[19] T.M. Shafaat, M. Moser, A. Ghodsi, S. Haridi, T. Schütt, and A. Reinefeld. Key-Based Consistency and Availability in Structured Overlay Networks. Third Intl. ICST Conference on Scalable Information Systems, June 2008.

[20] I. Stoica, R. Morris, M.F. Kaashoek D. Karger, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet application. *ACM SIGCOMM 2001*, Aug. 2001.