Project no. IST-033576

XtreemOS

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

Extended version of a scalable publish/subscribe system

D3.2.12

Due date of deliverable: November 30^{th}, 2009
Actual submission date: Dec 7^{th}, 2009

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: Christian Hennig
Zuse Institute Berlin
Takustrasse 7
14195 Berlin
Germany

Version 1.0 / Last edited by Thorsten Schütt / Dec 4^{th}, 2009

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	2009/10/16	Christian Hennig	ZIB	first draft
0.2	2009/10/20	Christian Hennig	ZIB	added benchmark results
0.3	2009/10/24	Christian Hennig	ZIB	clean-up
0.4	2009/10/27	Christian Hennig	ZIB	applied reviewer suggestions
0.5	2009/11/13	Thorsten Schütt	ZIB	applied reviewer suggestions
1.0	2009/12/04	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",it also includes an overview of new features for overlay maintenance and the results of a massive node failures experiment.

This system is a key component of the highly available and scalable infrastructure as described in deliverable D3.2.7 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 an 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. The transactions allow us to consistently update all replicas belonging to one item and to update several items in one atomic operation at the same time. The transaction framework employs Paxos.

The final layer is the "Publish/Subscribe System", which uses the layers below for managing subscribers and topics. In comparison to the results of the last year's deliverable, the subscribe performance has improved by 260% and the publish performance by 365%.

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. 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 short overview of our "Publish/Subscribe System" called Scalaris in Sec. 2. In Sec. 3 we will explain the overlay maintenance, this is fundamental to build a robustness service. In Sec. 4, we run several tests with different numbers of nodes to evaluate the system's scalability. We will show that the Pub/Sub system tolerant of node failures and scales linearly with the number of nodes.

2 System Description

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

As a common key aspect, all these 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, subscribing to topics, 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

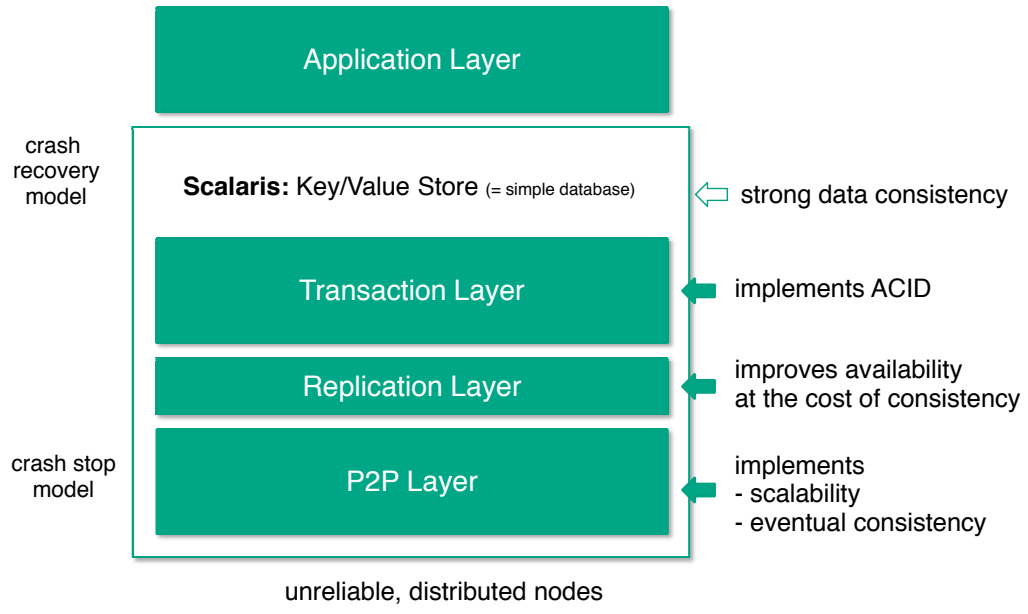


Figure 1: Scalaris system architecture.

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 target 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 that is necessary for low latency access in multi-datacenter environments.

- The middle layer implements data replication. It ensures the availability of data even under harsh conditions such as frequent node crashes and physical network failures.

- The top layer provides 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.

2.2 P2P Overlay

At the bottom layer, we use the structured overlay protocol Chord$^{\#}$ [9, 10] 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 [11], 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 [9].

2.3 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-management 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 optimizing:* 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 datacenters, these algorithms are used to place frequently accessed items nearby the users.

- *Self tuning:* Scalaris includes several timers to trigger internal management tasks, e.q. routing table maintenance. These intervals are tuned by taking into account environmental parameters like churn. Scalaris decreases the effort for system management in a stable environment like a datacenter, and increases it in an unstable environment like the internet. Therefore it also reduces human intervention.

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.

2.4 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 easily introduces deadlocks or lifelocks.

In the literature [4], the *actor model* [5] 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.

Since the last delivery we did a complete code review. That means we formalized the design pattern for all actors using a component driven approach. All actors have uniform efficient API and standard behavior.

As illustrated in Fig. 2, Scalaris uses six main components. The *overlay maintenance* uses our version of T-MAN and needs *cyclon* and the *dead node cache*. The *transaction manager* offers an interface for the "Publish/Subscribe System" and the *failure detector* informs the components of node failure. Our Erlang implementation of Scalaris comprises many components. It has a total of 16,000 lines of code: 12,000 for the P2P layer with replication and basic system infrastructure, and 2,700 lines for the transaction layer.

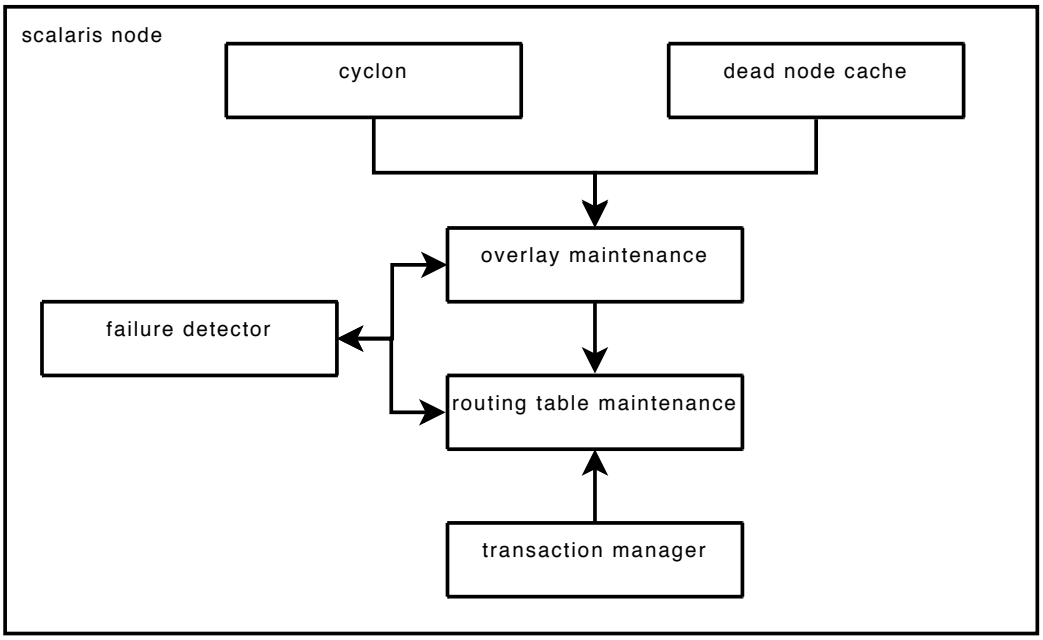


Figure 2: Scalaris components architecture.

3 Overlay maintenance

Scalaris at first used Chord's original ring overlay maintenance. These capabilities are insufficient for our scenario. There are some situations where Chord's ring overlay maintenance is not able to fix the ring and does not guarantee eventual consistency. These are correlated node failures, partitioning, repartitioning and loopy rings. Also the required time for fixing inconsistencies of Chord's overlay maintenance is too long. Therefore we switched to T-MAN [6] + Cyclon [8]. Gossip-based ring maintenance protocols like T-MAN guarantee eventual consistency. As long as the underlying peer sampling algorithm continues to provide random nodes from the complete overlay, the ring will always be eventually fixed. For our experiments in Sec. 3.2, we used Cyclon as a peer sampling algorithm. It is reasonably robust in face of churn. Only when the Cyclon network becomes disconnected, the ring structure can break.

3.1 Gossip-based Ring Maintenance

T-MAN is a protocol for constructing overlay topologies with the help of ranking functions. The goal topology is derived in a decentralized manner by using local knowledge only. The algorithm needs several rounds until it converges to the goal

topology. Each node keeps a list of peers, the *view*, which are closest according to a given distance metric. Nodes regularly update their views by exchanging and merging them with other peers.

We adapted T-MAN for continuous ring maintenance:

- For data consistency, it is important to detect dead (crashed) nodes as soon as possible. To accelerate the detection, each node in the local view is watched by a failure detector. In case of a failure, the node is removed from the view.

- Dead nodes are stored in a *dead-node-cache (DNC)*. This is a FIFO queue with a fixed size of 10 elements in our case. The nodes in the DNC are periodically contacted to detect re-appearing nodes, e.g. after the repair of a network partitioning.

- T-MAN [3] initializes the local view with a set of random nodes. The shuffling continues until the view does not change anymore. When the view becomes stable, the view is re-initialized with random nodes and the procedure starts from the beginning. With this scheme, it can take up to $O(\log N)$ shuffle rounds until defects in the overlay are repaired.

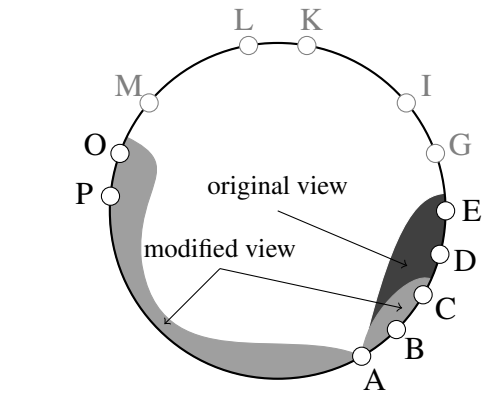
  In contrast to the original T-MAN, we never reset the local view and we include in each shuffle operation some random nodes.

- T-MAN uses a ranking function based on the distance in the key space, $d(a,b) = min(N - |a-b|, |a-b|)$, for building rings. Our view, in contrast, builds on predecessor and successor lists, as Fig. 3 shows. With a view size k we include the $k/2$ closest predecessors and successors in the view. This ensures that the nodes are always evenly balanced between predecessors and successors, even with unevenly distributed nodes. This improves the reliability under churn and with correlated failures.

3.2 Overlay maintenance Evaluation

We implemented the described algorithm with a ring-based overlay. For the experiment we simulated 400 nodes on two servers. 300 nodes were hosted on the first server ("left partition") and 100 nodes were hosted on the second server ("right partition"). The partitioning was simulated by removing the network connection between the two servers. We used the following system parameters:

Cyclon Interval	4.9s
T-MAN Interval	10s
Failure Detector Timeout	3s



original T-MAN: $view_A = [B, C, D, E]$
modified T-MAN: $view_A = [B, P, C, O]$

Figure 3: Succ/pred list with original resp. modified T-MAN.

We simulated the scenario where each server owns a disjoint segment of the ring. When partitioning the network, a contiguous segment of $1/4$ resp. $3/4$ nodes disappears.

Fig. 4 shows, for the left and the right partition, the size and the healthiness of the ring over the observation time (600s). 'Healthiness' is defined as follows: The nodes' positions in the succ/pred list of node n are based on a distance metric in the node space using knowledge ($\tilde{d}(x, y)$). Node n assumes that the first node in its succ/pred list ($succ$ and $pred$) has a distance of 1 in the node space ($\tilde{d}(n, succ) = 1$ and $\tilde{d}(n, pred) = 1$). For the healthiness metric, we sum up over all nodes in all views the differences between $\tilde{d}(x, y)$ and the distance based on global knowledge ($d(x, y)$). Additionally, we weight the errors according to their relative position in the list. It is more important for the direct successor to be correct than for the last node in the list. Finally, we normalize the healthiness to the interval $[0, 1)$.

At $t = 0s$, the system was started and 400 nodes joined the system, with the nodes' startup being staggered over the first 10s. Increasing the ring size from 1 to 400 within 10s causes a large amount of churn and T-MAN is not able to keep the succ/pred list correct. After ≈ 140 seconds, T-MAN has fixed the ring structure. In this period, T-MAN performed ≈ 14 shuffle rounds.

At $t = 200s$, we disconnected the two servers. From here on the views of the left and the right partition differ because they observe different networks. In the left partition, the ring size drops to 300 nodes, because the right partition disappeared and vice versa. At the same time, the error jumps up, because the pred/succ list of the nodes at the datacenters' borders point to dead nodes. After

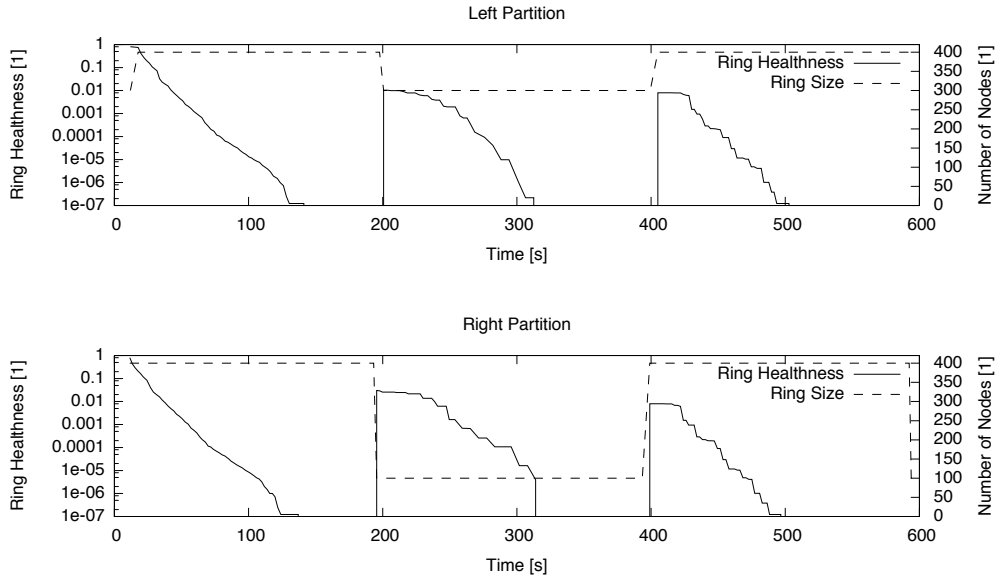


Figure 4: Network partitioning and repair with modified T-MAN (0s: startup, 200s: network partitioning, 400s: network repair)

$\approx 110s$ (or 11 shuffle rounds) the error drops to 0 and the system consists of two independent rings.

At $t = 400s$, we re-connected the links between the two servers and the nodes in the DNC are detected to have become alive again. Cyclon injects nodes from the other partition and T-MAN starts to repair the ring. As can be seen, the ring size goes up to 400 nodes and the ring becomes fixed after $\approx 100s$ (10 shuffle rounds).

Note that this repair procedure is possible in Chord. When all nodes in an arbitrary node's successor list are dead, the ring will be broken and cannot be fixed by Chord.

4 Performance Measurements

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 Dual-CPU Xeon 5150s (2 Cores per CPU) running at 2.66GHz and 8GB memory. The nodes are connected via GigE. All tests were performed on this GigE network. Revision 397 of Scalaris [7] and Erlang R13B01 were used. For the measurements 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 Scalaris nodes in each VM
3. Each threads executes the operation to test i times
4. Wait for all threads to finish
5. Stop stop watch

4.1 Field Trial of Scalaris

It is possible to start several configurations of scalaris on a cluster. We have to decide the number of ErlangVM per cluster node, the number of Scalaris nodes per cluster node and the number of clients. To figure out the best setup, we iterate over thes parameters. Therefore we set benchmarks with 1, 2 and 4 ErlangVMs per cluster node, with 4, 16 and 32 scalaris nodes per cluster node. These parameters where verified on 1, 2, 5, 10 and 15 cluster nodes. The load was generate either by 1, 2, 5, 10, 50, 100 or 200 clients per cluster node.

4.1.1 Cluster nodes: 1

The top 5 results for publish by using one cluster node and the corresponding configuration:

VMs/ Server	Nodes/ Server	Clients/ Server	Iterations/ Server	Publish/s	Subscribe/s
1	4	200	20000.0	25299.3	3224.9
1	4	10	20000.0	20883.2	3883.3
1	4	5	20000.0	20432.4	3448.8
1	4	50	20000.0	17544.6	3615.2
1	4	100	20000.0	17236.8	3483.4

The same for subscribe:

VMs/ Server	Nodes/ Server	Clients/ Server	Iterations/ Server	Publish/s	Subscribe/s
1	4	10	20000.0	20883.2	3883.3
1	4	50	20000.0	17544.6	3615.2
1	4	100	20000.0	17236.8	3483.4
1	4	5	20000.0	20432.4	3448.8
1	4	200	20000.0	25299.3	3224.9

4.1.2 Cluster nodes: 2

Results for publish by using two cluster nodes:

VMs/ Server	Nodes/ Server	Clients/ Server	Iterations/ Server	Publish/s	Subscribe/s
1	16	50	20000.0	17655.4	1235.3
4	16	50	20000.0	17473.6	2273.9
4	16	100	20000.0	17388.6	1373.1
4	16	200	20000.0	17298.0	1413.0
4	32	100	20000.0	15951.6	1169.4

The same for subscribe:

VMs/ Server	Nodes/ Server	Clients/ Server	Iterations/ Server	Publish/s	Subscribe/s
4	32	50	20000.0	15027.1	2630.9
4	4	200	20000.0	13010.1	2578.4
4	4	50	20000.0	13460.7	2454.0
2	32	100	20000.0	15857.0	2441.7
2	4	100	20000.0	14485.8	2427.1

4.1.3 Cluster nodes: 5

Results for publish by using five cluster nodes:

VMs/ Server	Nodes/ Server	Clients/ Server	Iterations/ Server	Publish/s	Subscribe/s
1	16	50	20000.0	37540.1	5694.4
4	4	200	20000.0	36784.1	4170.8
1	4	50	20000.0	35146.5	3147.4
4	4	50	20000.0	35094.5	4221.8
4	4	100	20000.0	34876.4	3457.0

The same for subscribe:

VMs/ Server	Nodes/ Server	Clients/ Server	Iterations/ Server	Publish/s	Subscribe/s
1	16	50	20000.0	37540.1	5694.4
4	16	200	20000.0	30758.5	5482.2
1	16	200	20000.0	33347.0	5476.5
4	32	200	20000.0	29135.8	5433.3
1	32	50	20000.0	26854.2	5125.0

4.1.4 Cluster nodes: 10

And the results for publish by using ten cluster nodes:

VMs/ Server	Nodes/ Server	Clients/ Server	Iterations/ Server	Publish/s	Subscribe/s
1	4	50	20000.0	61835.0	5099.3
4	4	200	20000.0	59736.4	7166.4
4	4	100	20000.0	56085.3	6746.0
1	4	100	20000.0	53712.3	6188.4
1	16	100	20000.0	53431.7	10167.9

The same for subscribe:

VMs/ Server	Nodes/ Server	Clients/ Server	Iterations/ Server	Publish/s	Subscribe/s
1	16	100	20000.0	53431.7	10167.9
1	32	100	20000.0	49276.9	9468.1
1	16	50	20000.0	47917.7	9309.1
1	32	50	20000.0	42279.1	9242.2
1	32	10	20000.0	39571.5	9198.3

4.1.5 Cluster nodes: 15

Results for publish by using 15 cluster nodes:

VMs/ Server	Nodes/ Server	Clients/ Server	Iterations/ Server	Publish/s	Subscribe/s
4	16	200	20000.0	73011.8	12497.1
1	16	50	20000.0	72062.1	13541.1
1	16	200	20000.0	72046.3	12954.0
1	32	100	20000.0	69961.3	13788.0
4	32	200	20000.0	69875.5	12255.2

The same for subscribe:

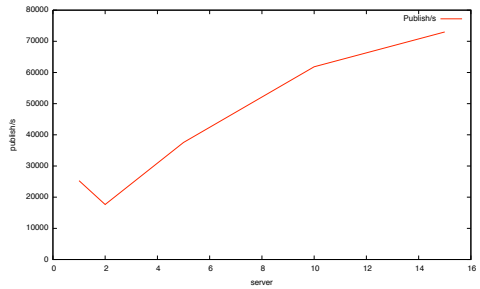
VMs/ Server	Nodes/ Server	Clients/ Server	Iterations/ Server	Publish/s	Subscribe/s
1	32	100	20000.0	69961.3	13788.0
1	16	50	20000.0	72062.1	13541.1
1	16	100	20000.0	63446.3	13024.5
1	16	200	20000.0	72046.3	12954.0
1	32	50	20000.0	64536.2	12765.9

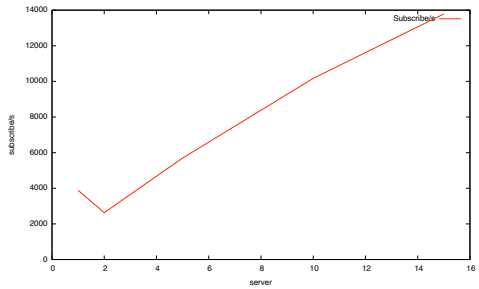
4.2 Summary of Results

Based on these results some rules were agreed upon for setting up Scalaris. At first, one ErlangVM is adequate for one machine. That was expected because version R13B01 handles SMP-systems much better than previous Erlang releases.

Secondly, the best performance was achieved by deploying 4 Scalaris nodes per CPU-core. Having more than one Scalaris node per core unsures that the process scheduler has enough runnable tasks to keep the cores busy. We also realized, Scalaris can handle a lot of clients at the same time. More than 200 clients per server are possible.

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





In comparison to the results of the last year's deliverable, the subscribe performance has improved by 260% and the publish performance by 365%.

5 Current State and Deployment

Since July 2008, Scalaris is available as open-source at [7]. We were contacted by several companies who expressed interest in using Scalaris. Therefore, there are users outside of XtreemOS. They give us a lot of hints and bug reports. That helped us improve scalaris, and to make it more convenient for productive use.

6 Conclusion

Scalaris provides a scalable and self managing pub/sub system and a transactional key-value store. Compared to other data services, Scalaris has significantly lower operating costs. Scalaris is now a robust and powerful service, that scales and is able to handle massive node failures.

References

[1] Anne-Marie Kermarrec and Maarten van Steen. Gossiping in distributed systems. *SIGOPS Oper. Syst. Rev.*, 41(5):2–7, 2007.

[2] Ayman Shaker and Douglas S. Reeves. Self-stabilizing structured ring topology p2p systems. *Peer-to-Peer Computing, IEEE International Conference on*, 0:39–46, 2005.

[3] A. Montresor, M. Jelasity, and O. Babaoglu. Chord on demand. In *Peer-to-Peer Computing, 2005. P2P 2005. Fifth IEEE International Conference on*, pages 87–94, Aug.-2 Sept. 2005.

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

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

[6] M. Jelasity and O. Babaoglu. T-Man: Gossip-based overlay topology management. ESOA, 2005.

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

[8] Spyros Voulgaris, Daniela Gavidia, and Maarten Steen. Cyclon: Inexpensive membership management for unstructured p2p overlays. *Journal of Network and Systems Management*, 13(2):197–217, June 2005.

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

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

[11] 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.