Building a framework for the consistency management of distributed applications

Vilmos Bilicki
University of Szeged
Department of Software Engineering
Hungary, 6720, Szeged
bilickiv@inf.u-szeged.hu

József Dániel Dombi
University of Szeged
Department of Software Engineering
Hungary, 6720, Szeged
dombijd@inf.u-szeged.hu

ABSTRACT

Distributed computing is leaving the laboratory and research lab environment and is now playing a significant role in the infrastructure of different companies and institutions. The requirements of running 7x24 without any noticeable failure can be effectively achieved only with a distributed architecture. The computing power and storage capacity of desktop machines have also become attractive as the basic building blocks of a distributed resource-sharing network.

Along with the useful properties of a distributed environment we get some challenges as well. A crucial question is that of consistent global knowledge among the distributed components. During the building and testing phases of our distributed software package called LanStore it turned out that currently there is no framework for .NET that offers group communication and consistency maintenance. There is the Peer-to-Peer API for unmanaged code that can be used in managed code, but this API was intended to be used in a WAN environment and it does not provide strong guarantees for consistency.

Hence we decided to design and build a framework that supports consistency management. One design criterion we applied was to support a highly changeable environment like that in a student computer laboratory. Our framework does not depend on any underlying communication infrastructure. It can provide the same set of services regardless of whether it is a peer-to-peer network or an IP level multicast network is used as the platform.

Keywords

Keywords: distributed system, consistency, group communication, peer-to-peer

1. INTRODUCTION

The number of the users with broadband Internet access is skyrocketing. According to estimates the number of users with broadband access in the U.S. increased by 36% in 2004. Now almost 70% of all U.S. home users have broadband connections. On a global scale, the number of the users in the world with Internet access grew by 182% during the period 2000-2005. 15.7% of the total world population now has Internet access. This penetration means that more than one billion users (one-sixth of the planet's human population) are connected to the Internet, which is probably the largest community on earth. The value of this community from the business perspective is constantly growing as well. The total Internet spending hit $143.2 billion in 2005[Eni05]. Yet the demands of this market differ from the conventional ones in several respects. The most important difference arises from the fact that, on the Internet, bank holidays and the different parts of the day lose their meaning. Business life should be run in a 7x24 way. But when this point is combined with the fact that the number of users who use a service is rather unpredictable, it is becomes clear that it is no easy task to develop such a system, one that is efficient, reliable and cost effective.

With the current high speed LAN and WAN network infrastructures the distributed paradigm is a reasonable solution for these problems. Such a service is provided by a group of processes that are operating and distributed throughout the network. The user should, however, see this system as a monolithic service and not notice its distributed nature. But using the network as a communication medium among processes introduces new problems. Current data networks - like IP networks - do not give guarantees for the correct delivery of the sent data. A developer has to take into account the variable aspects of the communication channel.

.NET Technologies 2006

One solution that has become more attractive is to use desktop machines as the basic building blocks of a distributed system. These PCs are less reliable than dedicated servers or they may run in environments where continuous operation is not guaranteed (as in a student laboratory for instance). Therefore a reliable distributed system should be able to tolerate the failing of one or more of its serving nodes. Depending on the type of tolerated failure, the system can become quite complex and costly to implement.

To overcome this complexity a common method is to use a framework that hides the failures of the system from the higher layers. Probably the biggest question for a distributed system is that of consistency. To be able to act as one virtual service the distributed system should have a consistent knowledge base. The message-oriented Group Communication Service (GCS) [Vit99] may provide the consistency for a distributed system.

There are well-known frameworks for providing the above-mentioned services, but we found just the Peer-to-Peer API [Win03] was available for the .NET environment. Our experience showed during the building and testing of the LanStore [Bil05] system that a well-tested, general, easily extendable consistency framework removes most of the burdens associated with testing and developing. Hence we decided to build the DCon framework to provide this functionality.

First we will introduce our new contribution, then we will outline the most common services available for group communication. One interesting approach is the Paxos algorithm, which will be evaluated in the next section, followed by a discussion of several well-know frameworks. As one of our goals was to build a framework for a student lab environment, in the next section we present the results of measurements that were conducted in our laboratories. Based on our measurements we designed a framework that is described in the implementation section. In the final section we draw some conclusions and suggest several possible directions for future study.

2. Our contribution

We carried out a set of a measurement to test the reliability of a typical campus computer laboratory. In the literature we found only the [Bol00] study about the reliability of the desktop machines, but this measurement was conducted on desktop machines used mainly by dedicated persons. In contrast, our measurements were conducted in a public student laboratory.

We decided to implement a distributed consistency management framework, we know this is the only distributed consistency management framework for the .NET environment. Our system can use the services of a peer-to-peer network and native IP level multicast too. We implemented the Paxos algorithm [Lam00, Lam01] in a way that is optimal for frequently changing networks (see measurements). Our Paxos implementation is able to handle the membership changes. We ported the Paxos algorithm to a Peer-to-Peer environment where the group members are not on a central list.

3. Distributed systems

A general distributed system may have an arbitrary number of components and each of these components may have a different task and a different state space but to the service user it behaves like a centralized monolithic system. These components may communicate in an arbitrary way. The fault tolerance of these components is usually solved by replication. The replicated components execute the same algorithm and each of them should have the same state. One popular approach is to model this system with state machines [Sch90]. A metric of a distributed system is the safety it provides. Here safety means the number and types of failures it survives without losing consistency. Another important metric is called liveness. This means that with different types and numbers of failures the distributed system can still progress. A widely used solution for the above mentioned issues is the view-oriented group communication service (GCS). Here service reliability is provided at the message level. The following basic services are defined:

1. Membership service
2. Reliable multicast

A view is a state of the system consisting of a set of active nodes. If this set changes, the view changes as well. The most important property provided by a GCS is called “Virtual Synchrony”. If two processes participate in the same two consecutive views the same set of message will be delivered. For further details the interested reader may peruse the article [Vit99].

4. Paxos

The “Virtual Synchrony” property provides the global ordering of the messages and a reliable message delivery in a distributed system. The price we pay for this solution is that it is not scalable. As was shown in the Spinglass article [Ken01], the systems providing “Virtual Synchrony” can scale effectively only up to several tens of nodes.

The classic Paxos [Lam00, Lam01] protocol solves the consensus problem for an asynchronous replicated system. It guarantees consistency in the case of benign failures. Hence this algorithm has better scalability properties than systems with the “Virtual Synchrony” property. The drawback is that the progress of the system is not guaranteed, and

the total order of messages is not fully controlled by the clients.

The algorithm solves the following problem. Let P be a set of processes and let V be the set of values. Every process in P can choose one value from the set V, the goal of the Paxos algorithm being to guarantee that only one from these selected values is accepted. The network can delay and multiply the messages arbitrarily; the participating nodes can crash and restart randomly but the Byzantine failures [Lam82] are not tolerated. In other cases system consistency is guaranteed. The progress of the system is guaranteed only in stable periods.

The functionality of Paxos is provided by two basic primitives: the quorum and a global order provider. The task of the quorum is to select at most one value from the available values. There are distributed solutions for preserving the global order of the messages (e.g. GCS), but sometimes a single decider can handle it more effectively. Paxos may be regarded as a special case of the view membership protocols [Lamps01]

5. Recent solutions

For handling the issues of a distributed system in the .NET environment one can use the P2P API [Win03] and the System.Transactions [Win06] namespace. P2P API provides a basic IP overlay infrastructure. As the consistency of the given reliable storage is based on timestamps and serials, and it does not give appropriate feedback about the success or failure of a transaction, it cannot be used in several critical services. The System.Transactions namespace in .Net 2.0 offers only classical transaction services. It is unsuitable for a consensus-based data consistency.

Group Communication Systems-based frameworks have a long history, and they are now in their fourth generation. Here we mention only the most well known frameworks.

Isis [Bir94] was the first and best-known primary component membership service. Among other services it defined and provided the "Virtual Synchrony" property for the first time.

Transis [Dal96] was the first GCS that utilised the native IP level multicast services. It was the first partitionable membership service. The system contains multicast clusters that are interconnected. It has a multicast flow control mechanism that controls the traffic at the network level. It also supports group communication. The messages can be unordered, causally ordered, and totally ordered and safely delivered.

Totem [Mos96] utilises the native IP multicast capabilities of the underlying network too. It provides a system-wide total ordering of the messages even in the case of network partition and remerge ("Extended Virtual Synchrony"). This goal is achieved with a logical ring where only the token holder may speak. In larger networks there are hierarchical ring topologies.

The goal of the Ensemble [Ken00] project was to improve the quality of the software used in the Isis project. Instead of the monolithic approach the system was implemented using modules and well-defined interfaces. The micro-protocol stack further improves the flexibility of the system. The code was implemented in the ML language, which is an O'Calm variant language. With this approach they were able to define and perform transformations on the code in a mathematically proven way.

Spinglass [Ken01] uses a revolutionary new approach. The currently used GCS's cannot be scaled up to a really large number of nodes. The Spinglass project addresses this problem and it uses "gossip-based" protocols to provide a highly scalable, secure and reliable Group Communication System. The gossip protocols emulate the spread of an infection in a crowded population. It employs a NNTP like protocol [Kan86] (Bimodal multicast) as the basic infrastructure provider. This protocol gives a steady data delivery rate with predictable, low variability in throughput. It provides only probabilistic guarantees of virtual synchrony.

6. Feasibility study

Our university has a computer science laboratory with 204 PCs. Students can either use the Windows or Linux operating systems from 8 am. to 8 pm., and they can switch between the operating systems whenever they want. We measured machine availability by pinging these machines every minute for 3 weeks between February 6 and February 25 in 2006. Based on the TTL value of the response we were able to detect not only the failures but the type of the operating system too.

We measured that a week the mean number of the online Windows workstations was always above the critical 50%.

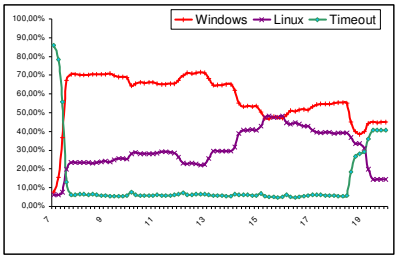

Figure 1: Operating system percentage / hours (2006.02.20)

The first figure shows the same statistics but now for a particular day. We notice that during the day except for a short period the number of online windows machines was above the critical level. The difference was about 10%. In the next figure the number of restarts is shown for another day. We

notice that there are situations where more than 10% of the machines are restarted. In such cases it may happen that during a transaction more than 50% of the windows machines are online but the ones that are running may vary.

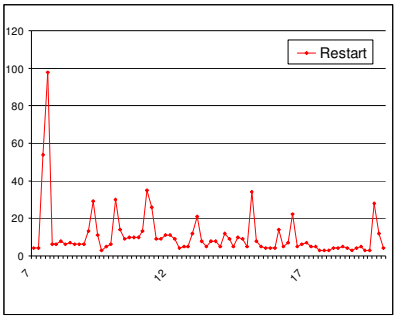

Figure 2: Number of restarts every 10 minutes (2006.02.14)

From these measurements we may conclude that for a reliable and liveness system we have to take into consideration these special time periods.

7. The DCon framework

The goal of the framework is to provide a distributed replicated data storage service with strong safety guarantees and weaker liveness properties. It can tolerate any arbitrary number of non-Byzantine failures. The liveness property is guaranteed only when more than the half of the nodes are active, but these nodes can change from time to time.

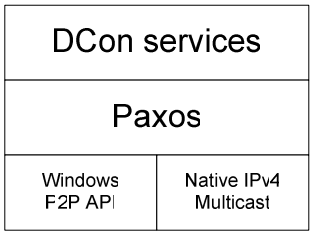

Figure 3.

We could have followed the approach of the above-mentioned frameworks and implemented a message-level GCS. But as our framework will provide only consistency services and not group communication services, we constructed it so that it would handle the issue of consistency more effectively. We selected the famous Paxos algorithm, which is ideally suited for these purposes. The reliability of this algorithm is mathematically proven. It can tolerate an arbitrary number of non-Byzantine failures without losing consistency. To be able to use it in a WAN environment and to be effective in a LAN environment we implemented it on the top of the Windows Peer-to-Peer API and the native IP level multicast services.

The DCon framework has three layers. These layers are shown in Figure 3. The first layer hides the distributed nature of the system from the user. It provides basic data manipulation and configuration services for the user. A data item can be added to the system, and existing data items can retrieved by a slow or fast query (see the next section). There are several methods available for reconfiguring the system.

The second layer implements the Paxos algorithm in a network independent way. At the bottom are the network dependent modules. Currently there are two modules: the native IP level multicast module and the module based on the services provided by the Windows Peer-to-Peer API.

In the following section we will describe our implementation of the Paxos algorithm in native multicast and P2P environments.

8. Our Paxos implementation

Functionality is provided by three abstractions: Leader, Consensus algorithm, Learner.

From a higher point of view the system works as follows. The clients send instructions to a leader. This leader carries out a three-phase transaction on the participating nodes and sends the results to the client.

Now we will describe the algorithm and a detailed description of our implementation (please consult Figure 4 for details).

Firstly, during the implementation phase of the classic Paxos algorithm we had to solve the following problems:

Message ordering: The purpose of the leader abstraction is to serialise the incoming requests. As we have seen this task can be done in a distributed manner (with logical timestamps and so on), but these solutions are more costly and are less reliable than the single leader solution. One could argue that the single leader incorporates a single point of failure into system. This is true, but as the leader does not have persistent data it can be easily replaced by a live substitute.

Leader election: As a communication medium between the Leader and the participating nodes, the *Instructions* multicast channel is used. During idle periods, the Leader periodically multicasts a beacon packet that contains the number label of the latest instruction. Based on our experience in other fields we chose to set this period to 10 seconds. During active periods these packets contain Paxos instructions (*Propose, Accept, Decide*). Failure detection is achieved by timeouts. If there is no traffic on this channel for three times the beacon period (30 seconds), the clients will submit a *LeaderSelect* frame that contains their stability properties (the greatest message serial known by this node, the number of restarts, the duration of the longest stable period). Each node compares the received values with its values and if it discovers

that its values were better (in the case of equal values the greater IP number is chosen) it will wait for a random period between 0 and 15 seconds and start sending beacon packets. If a node thinks that it has the best values but receives beacon packets it will accept the new leader. With these settings a Leader change will last at most 45 seconds.

Learning the actual leader: There is a *Leader* channel where the leader submits the beacon every 30 seconds. This channel is intended for clients for them to determine the actual leader. The clients send the data items to be stored to the leader using a TCP connection.

Monotony maintenance: The leader node retransmits the messages from the clients to the *Instructions* multicast channel and these values assigns a global number *G* and local number *N* to the messages. Local numbers are interesting only when there are two or more leaders. These numbers should be unique among the leaders so it is constructed as follows: IP address+ $N*2^{32}$. For every submitted message *G* is increased and *N* is reset to 0. *G* and *N* are included in the beacon packets as well.

Every node in the distributed system is subscribed to the *Instructions* multicast channel. For every different global number there will be a separate "Synod" protocol that guarantees consistency among the nodes. It works as follows:

Phase 1. The leader selects a global *G* and a local number *N* for the instruction and sends it as a proposal for the nodes subscribed to the *Instructions* channel. This is the so-called *Prepare* request. If a receiving node receives a *Prepare* request it checks whether it is able to accept it. If the last accepted request has a global number which equals the received global number and the local number is less than that of the current request then it responds with a reject answer, otherwise it will send a prepare accept response. Both of the responses contain the last accepted request and the also the number of this request.

Phase 2. If the leader receives a response for its propose request from the majority of the nodes, then it selects the latest accepted request, or if there was no request previously then it uses its own request and sends an accept request to the *Instructions* channel. In the case of insufficient responses or a reject answer it will increase the local number and submit the prepare request again. If there are insufficient responses after the fifth unsuccessful round it will stop the process and send an unsuccessful message to the clients. If it gets one or more reject answers it will increase the *N* value and send the message again. After five unsuccessful turns it will increase the value of *G* to the maximal value reported by the clients plus one received in the reject messages. If it is unsuccessful then it will report this to the client. This situation can happen only when there are several leaders and all are functioning for a longer period of time. But this may happen only in very special circumstances. It is quite rare.

The node receiving an accept request checks the local number of the request, and if it is greater than the last accepted one or there was no such *G* then it accepts the request and sends an accepted message to the leader. Otherwise a reject response is sent with the *N* value and maximal known *G* value.

Phase 3. After receiving sufficient accepted messages the Leader sends a *Decided* message to the *Instructions* multicast channel. The node that receives the *Decided* message will insert the Decided values into its Decided values storage. The timeout for each phase is 20 seconds. If the number of received accept messages was less than the previously defined majority value it will try sending the accept request again. If it fails five times it will send this result to the client and stop the process. In the case of a reject message it will follow the process described in Phase 2 and restart Phase 1. If the chosen value was not the value originally sent by client, then the Leader will repeat the whole process until the decided value and the accepted value coincide. This situation may occur if the *G* known by the leader is less than the greatest *G* in the whole system.

A detailed description of this algorithm can be found in [Lam00, Lam01]. We implemented the Paxos algorithm using several optimisations to achieve better response times:

For the system to progress we need the majority of nodes to be live. It may happen that in a fluctuating system, the majority of nodes are always present but are constantly changing. For example the prepare request is received by node A, then node A restarts and node B finishes its restarting process. So node B will only receive an accept request. The classic Paxos algorithm recommends rejecting this message. But with this solution it can happen that we have to replay the whole propose/accept procedure. Instead of this we suggest the following. If a node receives an accept request without previously receiving a propose request it shall answer this request. If it disagrees with the value suggested by the accept request it shall handle the accept request as a propose request; if it agrees with the received value then it shall handle the accept request as a propose and accept request. With this modification we did not change the durability of the algorithm, but in some cases we reduced the required number of message exchange from six to two. This algorithm is described in [Lam01].

```
Phase1:
Server:
        Var ReceivedRequest([G[N,V]], Iteration=0
        SendPropose(Nx2^32,G)
```

```
Node:
        Var ReceivedProposes [G[S,V]]
        ReceivePropose(S,G){
        IF(G not known)
                SendAcceptPropose()
        ELSE IF (Smax <S)
                SendAcceptPropose(Sloc,Svalue)
        ELSE
                SendRejectPropose(G,Smax,Gmax)
        }
Phase2:
Server:
        IF (ReceivedAcceptPropose > Memb/2)
                IF(MAX(S) != 0)
                        SendAcceptReq(G,Sloc,Svalue)
                ELSE
                        SendAcceptReq(G,Sloc,V)
        ELSE
                IF(N<5)
                        N=N+1
                        GOTO Phase1.
                ELSE
                        REPORT ERROR
Node:
        IF(G not known)
                SendAcceptReq()
        ELSE IF (Smax <S)
                SendAcceptReq ()
        ELSE
                SendRejectReq(G,Smax,Gmax)
Phase3:
Server:
        IF NUM(ReceivedAcceptReq > Memb/2)
                SendDecide(G,V)
        ELSE
                IF(N<5)
                        N = N+1
                        GOTO Phase 1
        IF(Sv != V)
                G = G+1
                GOTO Phase 1
        ELSE
                SendSuccess()
```

Figure 4. Algorithm

Change of membership: The participating nodes maintain two lists of instructions. In the "Client list" are stored the data items submitted by the clients, while the "System list" contains the instructions for system maintenance. The handling change of membership is solved by these special instructions, which are treated the same way as instructions from clients.

Message optimization: A *Leader* may incorporate an arbitrary number of Paxos messages with different *G* values into one submitted packet. The Decide packets may be piggybacked to Accept packets. The prepare packets are only needed during the start of a longer stable period. With these optimisations we then need only one message per transaction during stable periods. The details of these optimisations were mentioned in part in two papers [Lam00, Lam01].

Slow/Fast query: A client may learn the chosen values in a fast or slow way. The fast way is to query the adjacent node about its list of decided values. The slow way is to perform a distributed query of the missing values. This query is submitted to the *Instructions* multicast channel. The distributed query contains the number label of the last known decision. The nodes receiving the query will respond to and return the accepted values. The client will summarise the answers and in the case of unknown new decisions it will send a decide message to the *Instructions* multicast channel to help the progress of the whole system.

9. Measurement

We tested our implementation in different circumstances to prove that the single leader role does not affect its stability.

To be able to simulate different network conditions we developed a simulation framework where every machine was simulated with separate thread. With the help of this solution we were able to fine tune the machine restart probabilities.

In the following we will present our results about the stability of the leader election process. During the experiment we simulated 200 PCs with the restarting probability of 10% to 50% . On the Figure 5 we can notice, that the system converged in a very fast manner in the case of low restarting probability. If we raise the restarting probability the system also converged, but in this case the convergence is slower, and it contains more peaks.

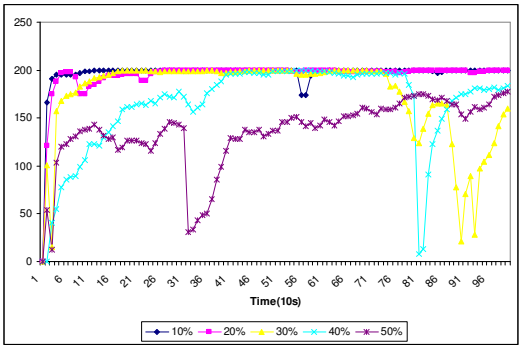

Figure 5. Number of threads, which are know the good leader at the same time

10. Paxos in a peer-to-peer environment

The services of the Windows Peer-to-Peer API were described in the recent solutions section. We can if we wish use it as a basic infrastructure to build an IP overlay multicast service. The communication service will be less efficient than in the native case, but in some situations we cannot use native multicast services anyway. The reliable storage service does not guarantee safety properties comparable to those of Paxos.

In our system we solved the following problems:

Group membership: To be able to implement a Paxos-like algorithm with guaranteed safety properties we have to know something about the success of the spread of the information. For this we need some membership details. As this framework assumes that there will be a high number of nodes there is no central information about the membership. To overcome this, we chose

to measure the total weight of the network and this value will be refined from time to time, but we will save only the maximal value while the system is running. Based on the maximal and the current value, the algorithm will be able to decide whether a partitioning has occurred and if this partition is capable of acting as a reliable storage medium. In the case of partitioning only the partition with weight more than the half of the whole weight will be suitable to act as a reliable storage. This solution works when, after the partitioning, there is a will on the user's part to merge the graph. If the partitioned sections start their lives separately, one can initiate a separate instance of the consistency algorithm on each. After doing so, it will be impossible to unify the network, however as the algorithm is intended to preserve global consistency there is no easy way of merging systems with a different history. With this membership view the nodes in the Peer-to-Peer network act as Paxos nodes.

Global order: This can be handled in a distributed or centralised way. The decentralised solution may be a more suitable solution for a peer-to-peer network, but as the Windows Peer-to-Peer API uses a central point of the network for graph maintenance we opted for this solution. A step toward the fully decentralised solution could be the use of per client root nodes. In this case an additional iteration is needed to evaluate the global order of the values. This could be done with the help of the weight of the groups which accepted a value with the same serial number.

After this high-level overview we will describe how our solution works:

The Peer-to-Peer network or segment has a central point- the node with the smallest ID (the same node being used for graph maintenance). This node sends a beacon signal every T seconds to each of its neighbours. The main task of this beacon is to measure the weight of the network.

Loop free message transfer: The graph constructed by the P2P system is a redundant one, hence we need an algorithm to avoid the situation of message loops. The P2PDatabase article [Awe02] advocates using spanning trees, but in a dynamic network it would be a costly solution. So we decided to use the well-known "Path Vector" algorithm [BGP06][Win03] (the same idea being used for name queries in MS P2P API). Every beacon packet has a path vector attribute that contains the sequence of nodes it traversed during its trip. If a node receives a beacon packet it first checks whether it is present in this attribute. If it finds its ID then the packet will be discarded. It then inserts its ID at the end of the path vector attribute and submits the packet to each of its neighbours except the neighbours which are present in the path vector. With this solution we have a multicast communication infrastructure.

Aggregated feedback: To measure the current weight of the network, each node will send a feedback to each beacon packet with the aggregate number of feedback packets received. Every non-leaf node (i.e. one which transmitted a beacon packet) has to wait for an answer for each submitted beacon packet. As we use the services of the Windows Peer-to-Peer API, theoretically the neighbours are always online (if not, the graph maintenance algorithm will correct this), but to avoid a potentially long delay of 5 minutes, every node has to maintain a timer for each submitted beacon frame. The timeout value will be inversely proportional to the number of nodes in the path vector attribute. In the case of a timeout it will send back a packet with a weight value of one. If there are redundant paths it may send the same feedback back several times. To avoid this, the synchronising packets contain a timestamp. A node will answer with an aggregate weight only for the first packet, and for the remaining packets with the same timestamp it will respond with a feedback containing a zero weight. Finally the root node will aggregate the feedbacks and this number will be the weight of the current network. The maximal value during this time will be the membership weight of the network. To ensure that this value is common knowledge, it will be attached to each beacon frame and stored at every node.

The root node acts as the leader in our Paxos algorithm. The algorithm is the same as in the case of native multicast, the only difference being that the nodes aggregate the answers they receive and send this answer as feedback values. The *Propose*, *Accept*, and *Decide* packets can act as beacon packets too. The root node will send beacon packets only after a defined idle time. To minimise the network traffic a submitted packet may contain several Paxos packets for several instructions and types. The root node will receive an aggregated feedback from participating nodes. The weight of the response should have a value greater than maxweight/2.

Slow/Fast query: The fast query option is the same as in the native multicast case. The slow query contains the last known decision number. The algorithm is the same as in the case of beacon packets. The feedback packets will contain the decisions known by the traversing nodes. Every transmitting node will check the feedback values for unknown decisions and then store them. If a node discovers that one or more of its accepted values are not present among the decided values, it will attach these values to the voted values. If it finds its accepted values among voted values, then it will increase the counter for these values. Thus

the client will be able to learn the decided values and also the values accepted by the majority of nodes. The detection of root node failure is handled by the underlying framework. Each node also checks whether it is the root node of the new graph. If it finds that it is, then it will initiate a query to learn the last synchronising number of the decided and proposed values. The result of this query will be the weight value of the current network. If it finds that it is larger than the half of the previous one, then it will start acting as the leader.

11. Conclusions and future work

In this article we described a solution which provides consistency services in a distributed environment. We implemented the well-known Paxos algorithm and solved several associated problems. As our framework handles only the consistency problem and it provides no group communication services ours should not really be compared to recent systems like Isis and Transis.

Our goal was to provide a simple and reliable API for consistency handling. Currently we also provide the same set of services on the P2P framework and on native IP level multicast.

Our software package is now in the development stage. Timing can be critical in a distributed system. The current values are based on our experience in the field of IP routing where the neighbour maintenance solves the same failure detection issue [OSPF96]. The tuning of the timeout values should be done in a real environment and software package should be tested under a variety of conditions.

In the future we would like to add a gossip-based module that can be deployed in the Windows P2P API. With this module the framework will not just be effective in LAN, but will be scalable in WAN as well.

12. Acknowledgement

We would like to thank David P. Curley for checking this article from a linguistic point of view.

REFERENCES

[Awe02] B. Awerbuch and C. Tutu. Maintaining Database Consistency in Peer to Peer Networks. Technical Report, CNDS-2002-2. 2002

[Bil05] V. Bilicki. LanStore: a highly distributed reliable file storage system, .NET Technologies'2005 conference proceedings, ISBN 80-86943-01-1, pp. 47-57, 2005

[Bir94] Birman K., van Renesse R. (editors) - Reliable Distributed Computing with the Isis Toolkit, IEEE Computer Society Press

[Bol00] W. J. Bolosky, J. R. Douceur, D. Ely, and M. Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop PCs. In Proceedings of SIGMETRICS, Santa Clara, CA, June 2000.

[BGP06] Y Rekhter, T. Li, S. Hares. A Border Gateway Protocol 4 (BGP-4). http://www.ietf.org/rfc/rfc4271.txt

[Dal96] D. Malki and Y. Amir and D. Dolev and S. Kramer. he Transis Approach to High Availability Cluster Communication. Communications of the ACM, 39(4):63--70, April 1996.

[Eni05] Enid Burns. Broadband: Online Retail Sales Grew in 2005. http://www.clickz.com/stats/sectors/retailing/article.php/3575456 , January 2006

[Kan86] B. Kantor, P. Lampsley. RFC 977 - Network News Transfer Protocol. 1986 http://www.faqs.org/rfcs/rfc977.html

[Ken00] Ken Birman, Robert Constable, Mark Hayden, Christopher Kreitz, Ohad Rodeh, Robbert van Renesse, Werner Vogels. Proc. of the DARPA Information Survivability Conference & Exposition (DISCEX '00), January 25-27 2000 in Hilton Head, South Carolina.

[Ken01] R. Shostack, and M. Pease. Kenneth P. Birman, Robbert van Renesse and Werner Vogels. Spinglass: Secure and Scalable Communication Tools for Mission-Critical Computing. International Survivability Conference and Exposition. DARPA DISCEX-2001, Anaheim, California, June 2001.

[Lam00] L. Lamport. Part time parliament. ACM Trans. on Computer Systems, 16(2), May 1998.

[Lam01]L. Lamport. Paxos made simple. ACM SIGACT News Distributed Computing Column, 32(4), December 2001.

[Lam82]L. Lamport, R. Shostack, and M. Pease. The Byzantine Generals Problem. ACM Transactions on Programming Languages and Systems, 4(3):382-401, 1982.

[Lamps01] B. W. Lampson. The ABCDs of Paxos. Principles of Distributed Computing, 2001.

[Mos96] L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia, and C. A. LingleyPapadopoulos. Totem: A fault-tolerant multicast group communication system. Communications of the ACM, 39(4):54--63, April 1996.

[OSPF96] Y. Moy. OSPF Version 2. http://www.ietf.org/rfc/rfc2328.txt

[Sch90] Fred B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: {A} Tutorial. ACM Computing Surveys, 22(4):299-314, 1990.

[Vit99] R. Vitenberg and I. Keidar and G. Chockler and D. Dolev. Group Communication Specifications: A Comprehensive Study. Tech. report CS99-31, Comp. Sci. Inst., The Hebrew University of Jerusalem and MIT Technical Report MIT-LCS-TR-790, Sep. 1999.

[Win03] Microsoft. Introduction to Windows Peer-to-Peer Networking. November 2005. http://www.microsoft.com/technet/prodtechnol/winxppro/deploy/p2pintro.mspx

[Win06] Microsoft. System.Transactions Namespace. 2006. http://msdn2.microsoft.com/en-us/library/system.transactions.aspx