UNIVERSITY OF CAMBRIDGE

University of Karlsruhe (TH)

A flexible and scalable peer-to-peer multicast application using Bamboo

Studienarbeit at the Institute of Telematics
Prof. Dr. M. Zitterbart
Department of Computer Science
University of Karlsruhe (TH)

and

Systems Research Group
Computer Laboratory
University of Cambridge

by

cand. inform.
Marcel Dischinger

Supervisors:

Prof. Dr. J. Crowcroft
Prof. Dr. M. Zitterbart
Dipl.-Inform. Uwe Walter

Date: June, 27th 2004

Systems Research Group Institute of Telematics

Contents

1. Introduction

The Internet is nowadays used by millions of people. Changing basic mechanisms of the Internet (the basic protocols,even just adding additional features to them) is not easy to accomplish, as this could mean changing the protocol implementation on every node.

Because of this overlay networks are very popular, especially in the scientific environment. Overlays offer a virtual network "over" the underlying topology, with their own routing schemes and often their own address space. They are situated at the application level of the TCP/IP model. Examples are Virtual Private Networks (VPNs), and many peer-to-peer systems using overlays.

Peer-to-peer overlays do not use centralized services and do not need any support from Internet Service Providers (ISPs). They help deploy novel required services like *multicast* and *anycast* in today's environment.

Group communication means the communication between several partners where each node acts as a sender, listener or both. The notion *multicast* stands for the communication of a single sender with multiple listeners (*1:n* or singlepoint-to-multipoint communication) whereas *multipeer* means a *n:m* relationship (or multi-point-to-multipoint communication) of senders to listeners. You want such a service for video conferences or online computer games. The idea behind these schemes is to aggregate several flows to a single flow, as long as they are taking the same path through the network to save bandwidth. *IP multicast* is the proposal to implement such a service in the IP layer of the Internet (see RFC988, RFC1054, RFC1112).

When speaking of *multicast* in this report it means the "any source multicast" and not the "single source multicast" model.

As native IP Multicast deployment is not very widespread, overlays allow a more independent way for implementing multicast services and applications using the existing internet infrastructure. Peer-to-peer overlays are especially popular in computer science research.

1.1 Goals of the project

The goal of this project is the implementation and evaluation of a scalable and flexible whiteboard multicast application based on the Bamboo peer-to-peer system.

The application should provide basic functionality like subscribing to an arbitrary topic and publishing changes on the canvas to the multicast group associated to the topic. The multicast groups are meant to be dynamic. That means that you can join and leave the groups at any time. The chosen multicast protocol should scale for large groups and be able to repair its multicast tree automatically due to changes in membership.

Because of the limited time to complete this project it was not a goal to provide any security mechanism for either the protocol or the whiteboard application, such as secure transmissions or access control.

1.2 Peer-to-peer systems

In general, a peer-to-peer computer network does not have fixed clients and servers but a number of peer nodes that function as both client and server to any other node of the network, so any node is able to initiate or complete any supported transaction.

Peer-to-peer is a class of applications that takes advantage of resources storage, cycles, content, human presence available at the edge of the Internet. Because accessing these decentralized resources means operating in an environment of unstable connectivity and unpredictable IP addresses, peer-to-peer nodes must operate outside the DNS system and have significant or total autonomy from central servers [Shir00].

Popular examples of peer-to-peer systems are instant messaging clients like *ICQ* or file sharing networks like *Gnutella* or *Napster* (beside Napster and ICQ use a client-server system for searching). Another example is the *SETI@Home* project which is a scientific experiment that uses Internet-connected computers in the search for extraterrestrial intelligence. But like Napster it is only partly a real peer-to-peer system: The computation is done peer-to-peer like, while the data distribution is done using a client-server-scheme.
The most common peer-to-peer system today is IP routing, by the way.

Peer-to-peer systems are a quite popular research field in computer science and are often used to realize completely decentralized networks of peers. In this terms there are applications for group communication, archival storage, content distribution and resource sharing. Popular examples are Chord [SMKK$^+$01], Pastry [CDHR02], [RoDr01] and Tapestry [ZHSR$^+$04].

In this work, I used a pretty new peer-to-peer system called Bamboo-DHT [Rhea04b].

1.3 Bamboo

Bamboo is a peer-to-peer system primarily written by Sean Rhea of UC Berkeley, but it is based heavily on the OceanStore [KBCC$^+$00] and the libasync [ZYDM$^+$03] projects. It is written in Java and is available as open source.

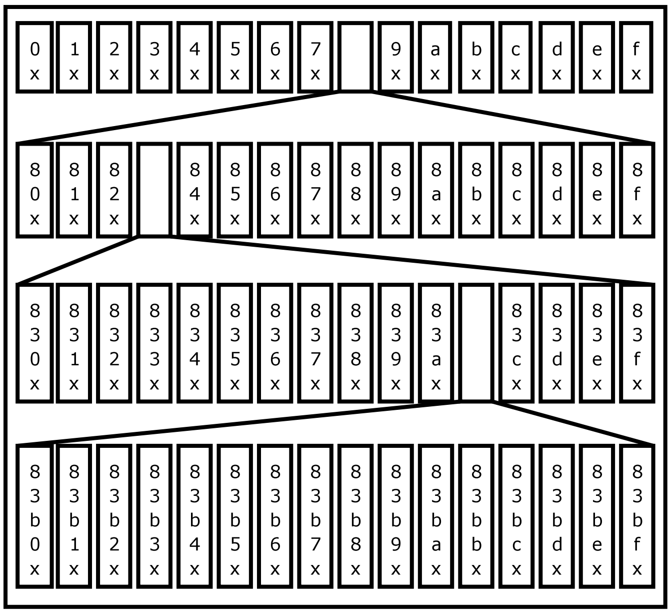


Figure 1.1: Routing table of a Bamboo node with node ID *83bx* and *b*=4. Digits are in base 16, *x* represents an arbitrary suffix

The development of Bamboo focused on a system that can handle high levels of *churn*. Churn is the continuous process of node arrival and departure. This is implemented by optimizing the bandwith usage which scales for Bamboo logarithmically the number of nodes. The main advantage of Bamboo compared to other popular scientific peer-to-peer systems like Pastry or Chord, is that it does not break down on high churn rates.

Bamboo itself is based on Pastry, but re-engineers its protocols. Compared to Pastry, the algorithms are more incremental, a difference that allows Bamboo to better withstand large membership changes as well as continuous churn in membership, especially in bandwidth-limited environments [RGRK03]. Beside of that it uses the same mechanism, such as the routing algorithm, and the join and leave behaviour.

Bamboo assigns to each of its node an unique, 160-bit node ID. The set of existing node IDs is uniformly distributed, achieved by using a secure hash (e.g. SHA-1) to generate it either of the IP address and port number or of a public key. A message addressed to a certain key is then reliably routed to the node with the node ID numerically closest to that key among all live nodes. Within a network of N nodes a message can be routed to any node in less than $\lceil \log_{2^b} N \rceil$ steps on average (with b a configurable parameter that effects the routing table size with respect to its granularity, and is typically 4).

Each node in the Bamboo network maintains a *leaf set* of *2k* nodes immediately proceding and following it in the circular identifier space with default size of 16 nodes (*k=8*). Additional, each Bamboo node maintains a *routing table*, too (see figure 1.1). Each node ID is treated as a sequence of digits of base 2^b denoting the entry *(l,i)* of the matrix the entry is chosen as a node whose identifier matches its own in exactly l digits and whose *(l+1)*th digit is i. The node closest to itself in network latency is picked from all nodes that can fill a certain routing table entry. The routing table is continuously improved by measuring the latency to neighbour nodes and updating the table with the new data.

Bamboo uses recursive routing to submit a message. Given a message with key D, routing is done by first calculating the longest matching prefix between D and the node's ID. Then it checks if the key lies within the leaf set, and if so forwards it to the numerically closest node. Routing terminates if this node is the local node itself (note that the local node is part of its own leaf set). If this fails it tries to find another node matching in a longer prefix than itself and forwards it to this node. If the search fails (because there is no entry of such a node) the message is forwarded to the numerical closest node of the leaf set.

Bamboo is written in an event-driven, single-threaded programming style [Rhea04a]. In first place it inherits its structure from SEDA, which stands for the *Staged, Event-Driven Architecture* [WeCB01]. As the name suggests, each part of a Bamboo application is a *stage*. Communication is done by passing events to every stage while the whole application is single-threaded. A stage registers itself to Bamboo and can also subscribe to events it want to receive and send messages to other stages or nodes. In the following I will use the notions "event" and "message" as synonyms.

1.4 Document outline

In chapter 2 I will describe the design of the implemented multicast protocol and the whiteboard application. Details of the implementation of these and the protocol mechanism in particular are described in chapter 3. In chapter 4 I will present the results of some experiments to analyse the behaviour of the implemented multicast protocol. Finally chapter 5 gives an outlook regards to the original goals, and possible future work and an overall conclusion of the whole project work.

2. Design

This chapter will give an overview of the design principles of the application, but will not deal with implementation details (you will find that in chapter 3).

This project can be divided into two parts: The multicast protocol layer and the whiteboard application. A main issue was to create as flexible a design as possible; therefore a very modular structure is needed. The advantage of this is that it is easy to replace a module by a different implementation, or to extend the whole application. For example, it is easy to replace the whiteboard with another application to just use the multicast layer as I have done in chapter 4 for evaluation.

As the Bamboo peer-to-peer system should be used, at least the multicast protocol layer must be implemented as a Bamboo stage. This stage must also provide a communication interface so that an application can use its services.

There are also several multicast protocols available like the reverse-path forwarding algorithm or Steiner minimal trees, for example. In fact there is no "best solution" for this kind of purpose at the moment. But the idea is to use the abilities of the Bamboo overlay network – like locality issues and routing – to build up a multicast tree.

2.1 The whiteboard application

A whiteboard application can have a lot of features. At least it must provide a canvas where the user can draw at and automatically publish these strokes to the multicast group. Furthermore it should be possible to switch the topic. When subscribing to a topic the current content of the canvas should be requested from another node and displayed.

Additional functionalities could be the provision of different colours to draw with, an erase function and the ability to insert text with the keyboard. Someone would also like to see a list of all currently subscribed nodes of a topic. You can think of plenty more features that might be useful to have with a whiteboard application.

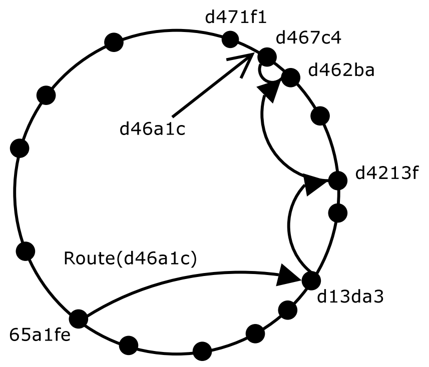


Figure 2.1: Routing a message from node *65a1fe* with key *d46a1c*.

2.2 The Bamboo API

The multicast part of this application will use the Bamboo peer-to-peer system for message sending and receiving. Therefore it is a good idea to get an overview of the supported API of Bamboo.

Bamboo exports a *message sink* to its stages. Messages can be dispatched to this sink in order to get sent. On the other hand applications can register their sink to Bamboo to receive messages. Furthermore – if you want to get messages routed by Bamboo – you have to register your application and get a unique appID that acts like a TCP/UDP port number, so that you can address a certain application on a Bamboo node.

In the following I will describe the provided message types for sending (and the according incoming messages), simplified for clarity.

Applications can use the following message types to send events:

- **BambooRouteInit(key, appID, msg)** if the application wishes to let Bamboo route that message to the node numerically closest to key and subscribed to the ID appID. A flag can be set to enable upcalls on all passing nodes.
- **BambooRouteContinue(msg)** is used to resend a message after an upcall.
- **NetworkMessage(address, msg)** can be used to send a message directly to an address. The transmission will fail if no node with the specified address is available.

The following messages are sent to the applications of a node by Bamboo:

- **BambooRouteDeliver(key, msg)** is sent by Bamboo if this node is the goal of a message (numerically closest to its key). It is only delivered to stages that match the appID.
- **BambooRouteUpcall(key, msg)** is sent by Bamboo if this node is on the way of the message and has a stage which matches the appID. The node can change the content of the message and have to resend it (if wished) using the BambooRouteContinue message.

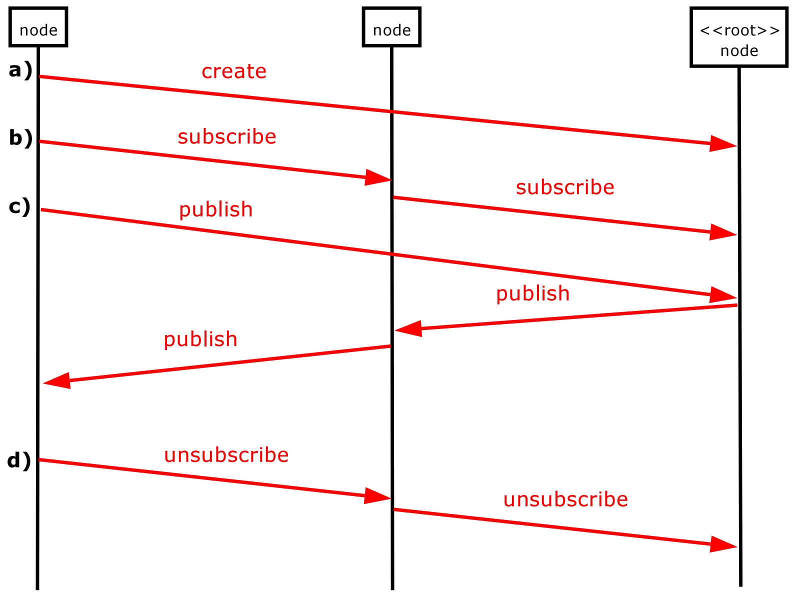


Figure 2.2: Sequence diagram of the multicast protocol.

- **NetworkMessage(msg)** is received when some node sent a message direct to a node without routing it through Bamboo.
- **NetworkMessageResult(success, returnObject)** is received for confirmation of a sent NetworkMessage and will contain a flag that informs about successful or unsuccessful transmission.

In figure 2.1 you can see an example for routing a message through Bamboo (by using the `BambooRouteInit` message type). The dots depict live nodes in Bamboo's circular namespace. The node IDs are shortened to 128 bit for simplification. If you would use a `NetworkMessage` to send this message it would fail, because there is no node alive with the specified key.

2.3 The multicast protocol

I chose to implement the same type of multicast protocol as Scribe – a reverse-path forwarding algorithm. Scribe is a multicast implementation for Pastry and is proofed to scale for large groups of nodes [RKCD02].

As this protocol is originally implemented using a peer-to-peer system it should be easy to port it to the Bamboo system. Especially as the underlaying system provides informations about locality which can be used easily to build a pretty good multicast tree.

The protocol offeres a simple API to its applications:

- **create(topicID)** creates a topic with name topicID.
- **subscribe(topicID)** causes the local node to subscribe to a topic with name topicID. All subsequently received messages for that topic are passed to that node and send to the local subscribed applications.

- **unsubscribe(topicID)** causes the local node to unsubscribe from a topic named topicID.
- **publish(topicID)** causes the message to be published to the topic representated by a topicID.

You can see a sequence diagram of the protocol in figure 2.2 that gives you an overview of the protocol flow which I will describe in the following briefly. A detailed description of the implemented API (specific in context of a Bamboo stage) can be found in section 3.4.

2.3.1 Topic management

Each topic has a unique topic ID which is associated with its own multicast group. Therefore I will use the notions topic and multicast group as synonyms. The topic ID is created from the textural name of the topic by using a collision resistant hash function (e.g. SHA-1) and is 160-bit long.

To *create* a topic (figure 2.2a), a CREATE message is routed through Bamboo with the topic ID as the key. Bamboo will deliver the message to the node numerically closest to the topic ID. The so chosen node adds the topic to its list of known topics and acts as from now as the root of the multicast tree for that topic. This node is furthermore the rendez-vous point for all published messages for this group.

2.3.2 Membership management

The tree itself is created using a scheme similar to reverse path forwarding [DaMe78]. A node which wants to *subscribe* to a topic (figure 2.2b) routes a SUBSCRIBE message towards the topics rendez-vous point with the topic ID as the key and upcalls enabled. So the message is routed by Bamboo and upcalls on every node which the message passes. These nodes then look up if they already know the topic and if so, the sender is added as a child and the message is terminated.
Otherwise a new topic structure is created – with the sender added as a child – and then inserted in the list of known topics. Afterwards the node routes a new SUBSCRIBE message towards the rendez-vous point (again with upcalls enabled) and tries to subscribe itself.

The message is terminated either when a node already knows the topic (and thus is already subscribed to it) or when it is delivered to the root itself (by a `BambooRouteDeliver` event then).

The *unsubscription* is done quite similar (figure 2.2d). The topic is marked as not locally subscribed any more. If there are no entries in the children table for that topic, a UNSUBSCRIBE message is sent to the parent node associated with this topic. The message then proceeds recursively up the multicast tree, until a node is reached that still has entries in the children table after removing the departing child or the root is reached.

The locality properties of Bamboo ensure that the network routes from the root to each subscriber are short with respect to the proximity metric. This is because Bamboo updates its routing table all the time by sending its own, special PING messages to measure the delay to the other nodes of the network.

2.3.3 Message publishing

If the publisher is aware of the rendez-vous point, it sends the PUBLISH message directly to it (figure 2.2c). The rendez-vous point sends the message to all its children then. The message proceeds recursively down the multicast tree, until a leaf node is reached. The caching of the rendez-vous point's address is an optimization, to avoid repeated routing through Bamboo, which is apparently slow compared to direct sending.

If the root of the multicast tree is not know by the publisher, it routes its message through Bamboo to that node. The rendez-vous point will send back an ACKNOWLEDGEMENT message to inform the publisher about its address.

As there is one rendez-vous point for every message, it is easy to add some access control to the protocol. By using a hash function that provides uniform distribution of the topic ID representation – and the same is true for the node ID of Bamboo – the roots of the topics will be uniformly distributed over all live nodes and the protocol will scale for large numbers of topics [RKCD02].

2.3.4 Repairing the multicast tree

Periodically, each non-leaf node in the tree sends a HEARTBEAT message to its children. These messages can be avoided if there are frequently published events. A child suspects that its parent is faulty when it fails to get HEARTBEAT messages for a configurable duration of time. In this case it will route a new SUBSCRIBE message through Bamboo to find a new parent.

If a child receives a heartbeat it sends back an ACKNOWLEDGEMENT message to its parent. The children tables are periodically refreshed by these messages. Children which fail to send an acknowledgement are discarded from the children table after some period.

If the rendez-vous point changes, then the old root can forward the publish message to the new one. If a root fails, the publisher can route to a new rendez-vous point through Bamboo. Also the recent root regulary sends DISCOVERY messages to search for a node numerically closer to the topicID than itself.

This tree repair mechanism scales well, because fault detection is done by sending messages only to a small number of nodes. Recovery from faults is local and only a small number of nodes is involved.

In the following chapter I will present in more detail the implementation of this protocol.

3. Implementation

The application should be implemented in Java, on one side because Bamboo itself is written in Java, on the other side to remain platform independent. The system will be deployed in a Linux environment. Because I am using the Java logging facility, at least version 1.4 of the JDK is needed to run the application.

To make the application as flexible as possible, I implemented the project in two independent parts. One is the multicast stage, called *ComCore*, the other the white-board application. As mentioned in chapter 2, at least the multicast part must be a Bamboo stage. In order to ease the communication between the two parts, I also implemented the whiteboard as a Bomboo stage, but with some slight differences which I will describe in section 3.3. In section 3.4 I will describe the details of the multicast protocol implementation.

But first I want to point out to some specialties of Bamboo you have to be aware of when you want to use it for application development.

3.1 A Bamboo stage

As Bamboo is a single-threaded, event-driven application you must be aware that every blocking or time-consuming process in a Bamboo stage will also block the whole event handler – means the whole Bamboo node. To prevent that it is necessary to use an own thread to wrap such code.

But using threaded code together with Bamboo causes other problems. Bamboo prevents the usage of its sending sink by other threads than the main thread. You have to use the `dispatch_later(message, time)` function in order to dispatch events from another thread than the Bamboo main thread, using a time value of 0. The normal `dispatch(message)` function is not working and will cause an exception. That is so because `dispatch_later()` ist the only thread-safe function in Bamboo and was made thread-safe just for this purpose [Rhea04a].

As mentioned before, your Bamboo application is a stage of Bamboo's event-driven architecture. If a new event occurs, it is passed sequentially to each registered stage.

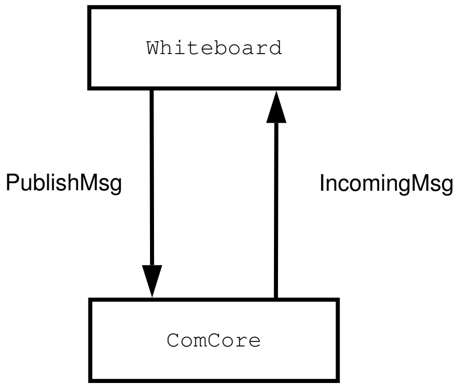


Figure 3.1: Local communication

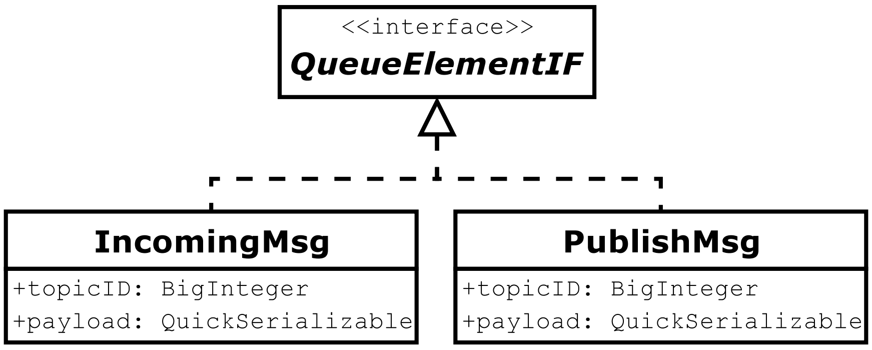


Figure 3.2: Locally used messages

So, a Bamboo application consists mainly of an `eventHandler(QueueElementIF event)` method that processes all incoming events [Disc04].

You can dispatch events only for the local node or for a distant node. Events can normally carry an arbitrary payload. Payloads must serialize their content into a buffer so that it can be sent. There are existing methods for serializing the basic Java types like `boolean, int` or `String`. Other objects must be converted to the supported types, otherwise they cannot be dispatched.

In order to receive an event, you have to register your event sink to Bamboo and subscribe for all the events you want to receive. If you also subscribe to the messages routed by Bamboo (see section 2.2) you need to request an application ID which acts like a TCP/UDP port. You will only receive Bamboo messages that are sent to your application ID. This is not true for all other types of events.

3.2 Messages

Before describing the details of the whiteboard application and the multicast layer I want to give an overview of the messaging system. This means the different message types and payloads that are used for special purposes.

3.2.1 Local messages

The local communication between the multicast layer and the application, which is using it, are done by using Bamboo to deliver them. As you can see in figure 3.1, there

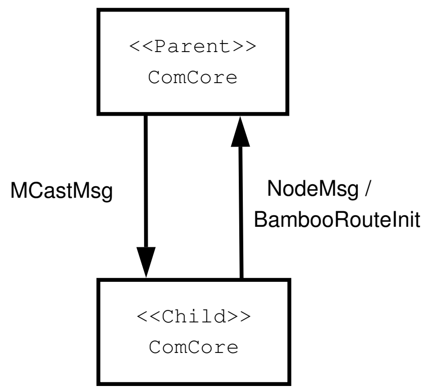


Figure 3.3: Communication between ComCore nodes

are two different messages, *PublishMsg* for messages from the application to the multicast layer and *IncomingMsg* for events from the mulitcast layer to an application. Both messages implement the `seda.sandStorm.api.QueueElementIF` interface so that they can be sent through Bamboo.

- **PublishMsg** is used to send messages to the multicast layer; e.g. containers with new strokes from the canvas or commands to subscribe or unsubscribe from a topic.
- **IncomingMsg** is used by the multicast layer to send events to all listening applications, e.g. containers with strokes from other whiteboards.

Each application that wants to use the multicast layer subscribes to `IncomingMsg` events. These events are broadcasted to all local listeners. Each of them has to sort out the messages it is subscribed to by the included topicID.

As you can see in figure 3.2, the two event types contain the topicID and a payload. How the communication works in detail will be described in the following sections.

3.2.2 Inter-node messages

In figure 3.3 you can see the two different types of communication between ComCore nodes: Up and down the tree. The main difference between these two directions is that we want the upgoing messages to be reliable while we not really care about the downgoing ones – as loosing a parent is serious, loosing a child is not a problem for a local node. The child itself has to take care that it is connected to the multicast tree.

For all messages going down the tree – normally messages from a parent to its children – ComCore uses `MCastMsg` messages. Its UML diagram is shown in figure 3.4. The message type `NetworkMessage` from which `MCastMsg` extends is not routed through Bamboo but sent directly to the specified address. You can choose if Bamboo should report back if the transmission was successful or not. This is turned off by default in `MCastMsg` messages.

If ComCore needs to send messages up the tree you have to differentiate between two possibilities: You know your goal or not. If you do not know your goal – e.g.

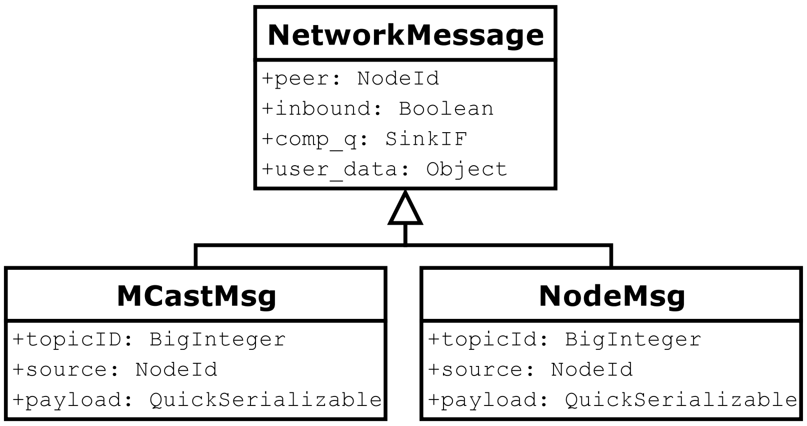


Figure 3.4: Message types for direct sending to an address

when you try to subscribe to a group or want to publish the first time – you have to use Bamboo to route the message by using a `BambooRouteInit` message. As routing through Bamboo is significant slower than direct sending you want to avoid that as often as possible. So the address of the parent and the address of the rendez-vous point of a topic is cached. `NodeMsg` messages are used to send messages directly up the tree. In this case you will get a message back if the transmission was successful or not and can react on that. This is because it is critical if we loose the rendez-vous point of our group or even our parent.

3.3 The whiteboard

The whiteboard should be a very simple application that uses the multicast layer. At this time it supports only basic features.

I divided the whiteboard application into two parts, one is the GUI and the interface to the multicast layer, the other is the canvas where you can draw at.

3.3.1 GUI and communication interface

The GUI is kept very simple, as you can see in figure 3.5. There is the canvas where you can draw at. At the right side are a few buttons to choose between draw- and erase-mode and to switch the topic. It is written using the Java Swing extension.

Bamboo is used for local communication with the multicast layer, but the whiteboard is subscribed only to `IncomingMsg` events. The inter-node communication is completely done by the multicast layer.

If one arrives, it checks the topic ID of the message and if it matches the topic that the whiteboard is subscribed to, it passes the included container to the canvas to draw it.

If there are new strokes drawn, the canvas calls the whiteboard and passes a container with vectors of the strokes. The container is then enveloped into a `PublishMsg` message and sent to the multicast layer. I will describe the containers in a while.

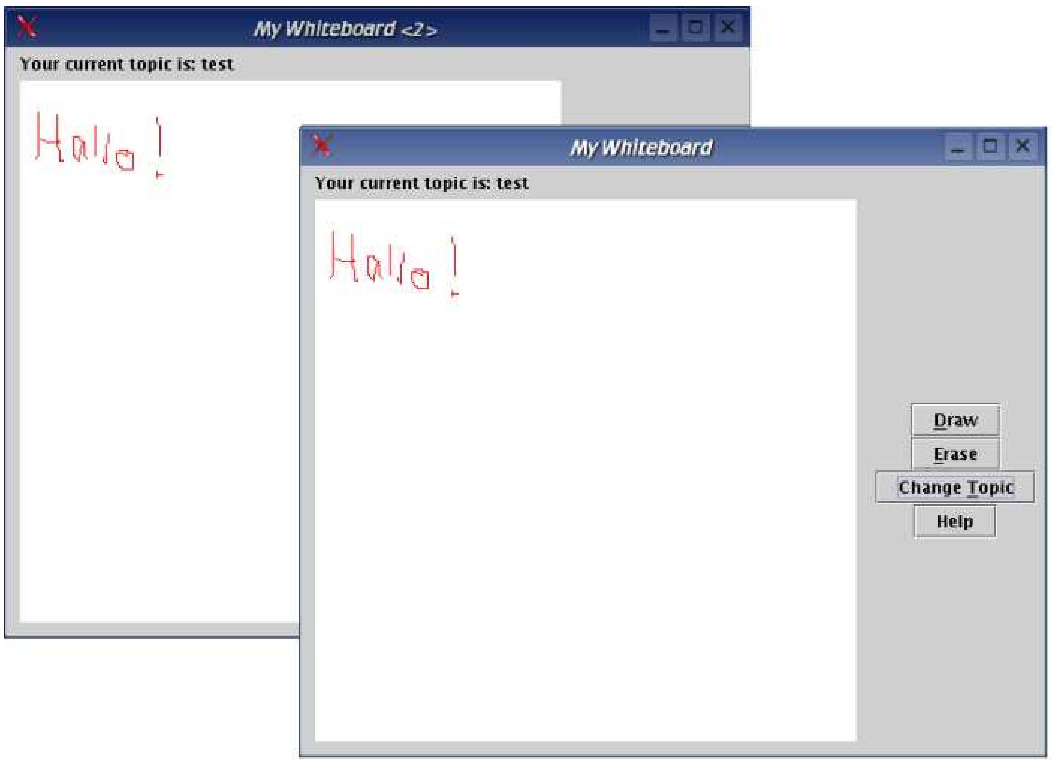


Figure 3.5: The GUI of the whiteboard

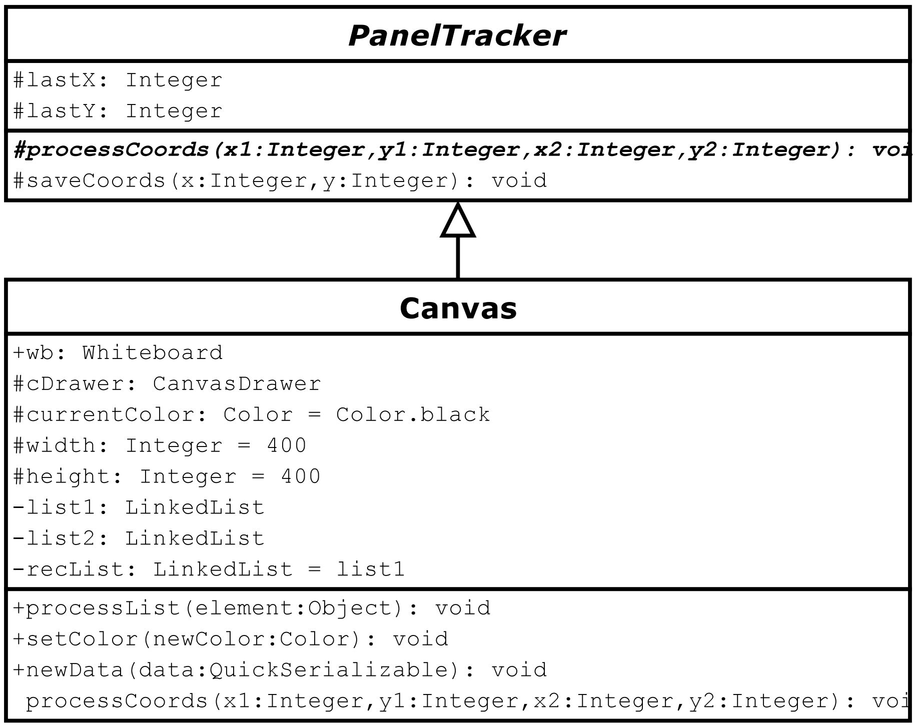

PanelTracker
#lastX: Integer
#lastY: Integer
#processCoords(x1:Integer,y1:Integer,x2:Integer,y2:Integer): voi
#saveCoords(x:Integer,y:Integer): void
Canvas
+wb: Whiteboard
#cDrawer: CanvasDrawer
#currentColor: Color = Color.black
#width: Integer = 400
#height: Integer = 400
-list1: LinkedList
-list2: LinkedList
-recList: LinkedList = list1
+processList(element:Object): void
+setColor(newColor:Color): void
+newData(data:QuickSerializable): void
processCoords(x1:Integer,y1:Integer,x2:Integer,y2:Integer): voi

Figure 3.6: Simplified UML diagram of the Canvas class

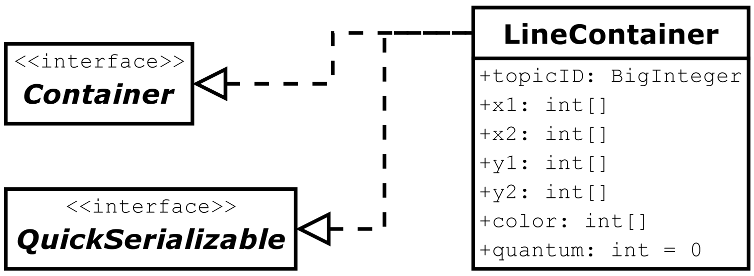


Figure 3.7: The LineContainer for sending multiple lines

3.3.2 The canvas

In figure 3.6 you can see a slightly simplified UML diagram of the *Canvas* class. It extends from the *PanelTracker* class which is meant to manage all user input. Within the Canvas class there are also two inline classes: One for drawing new strokes to the canvas and one for managing to send new strokes to the network.

If new data arrives for the whiteboard it calls the `newData()` function of the canvas and passes the container with the new data. In order not to block the whiteboard (and with it the whole Bamboo node) a new thread is started to process that data and to draw it on the canvas finally.

3.3.2.1 The PanelTracker

The `PanelTracker` class encapsulates all event listeners, e.g. the mouse listeners or keyboard listeners. All user input is processed by this class. It saves the last position in its variables `lastX` and `lastY` by calling the function `saveCoords(int x, int y)`.

For mouse input the according listeners call the abstract method `processCoords()` which must be implemented by an extending class. Its parameters are the start and the end of a line, which are saved to reproduce and draw the line.

So far, only mouse drawing is supported. But it is easy to extend this class to process keyboard input, for example.

3.3.2.2 Local data processing: CanvasDrawer

To support a smooth maintainance of the canvas another thread is started, called *CanvasDrawer*. It has two tasks: Make sure that the input is drawn to the canvas and that the local input is published to the network.

The core of the thread is an inifinite loop. To save CPU time the thread sleeps half a second within each cylce and tries to do some work then. This sleep is interrupted if there are new lines to draw.

The locally drawn lines are saved in a linked list. When this list extends more than 15 saved lines they are sent and the list is cleared. Anyway the lines in the list are sent after half a second. The list must be synchronized as the input of new lines and the sending of them are done by different threads. I chose to use two lists and to swap them if the recent list has to be cleared. The swapping and inserting is done by the synchronized method `processList()`.

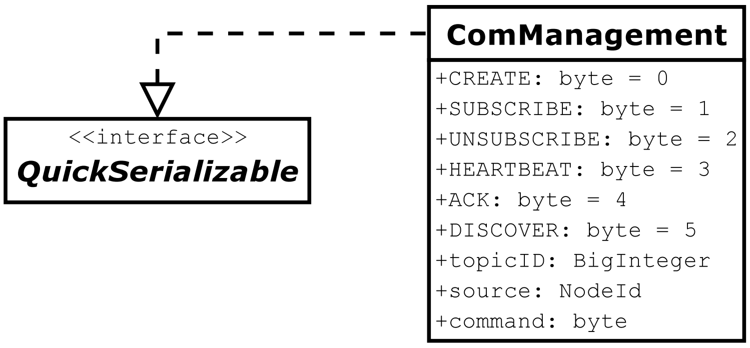


Figure 3.8: UML diagram of the ComManagement class

If data is ready for sending it has to be packed into a container (see figure 3.7. The container must be of type `QuickSerializable` so that it can be sent through Bamboo. Also it has to implement the empty interface `Container` to be accepted by the multicast layer. The usage of this interface allows transparent sending of data from application to application over the multicast layer.

The `LineContainer` supports packing of multiple lines with a color value for each. The start and end points of the lines and the color values are saved in simple integer arrays. The variable `quantum` saves how many lines are stored in the container.

The container is then passed to the Whiteboard class for publishing.

3.4 The multicast stage: ComCore

The multicast layer is implemented as a full Bamboo stage. It is transparent to upper layers (applications) and provides multicast service for multiple groups. After the more formal description of the algorithm in section 2.3 I will now point to specific issues of the implementation here.

3.4.1 The ComManagement messages

To maintain the multicast groups the `ComManagement` class is used for communication (see figure 3.8). It provides fields for the concerned topic (the multicast group respectively), for the source of this message and for a command. The supported commands are added as constant variables: CREATE, SUBSCRIBE, UNSUBSCRIBE, HEARTBEAT, ACK, DISCOVER, NEWROOT.

The commands CREATE, SUBSCRIBE, UNSUBSCRIBE are also used by the applications using ComCore to create, subscribe and unsubscribe a topic. I want to give here a brief overview:

- CREATE is used to create a new multicast group.
- SUBSCRIBE is used to subscribe to a multicast group.
- UNSUBSCRIBE is used to unsubscribe from a multicast group.
- HEARTBEAT is sent by a parent to its children to give a life sign.
- ACK is sent by the children to their parent to give a life sign.

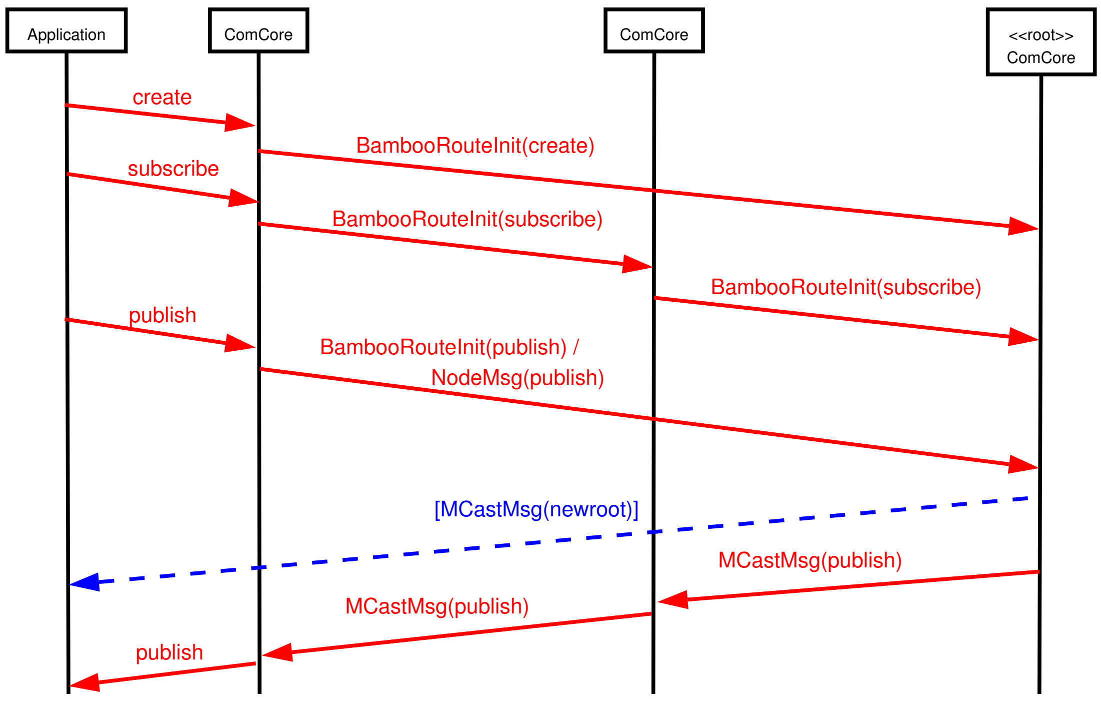


Figure 3.9: Sequence diagram of the protocol

- DISCOVER is sent by the root of a multicast group to discover if there is a node numerically closer to the topic ID then itself and therefore should become root.
- NEWROOT is sent when the root of a multicast tree wants to inform someone of its address.

3.4.2 Protocol implementation

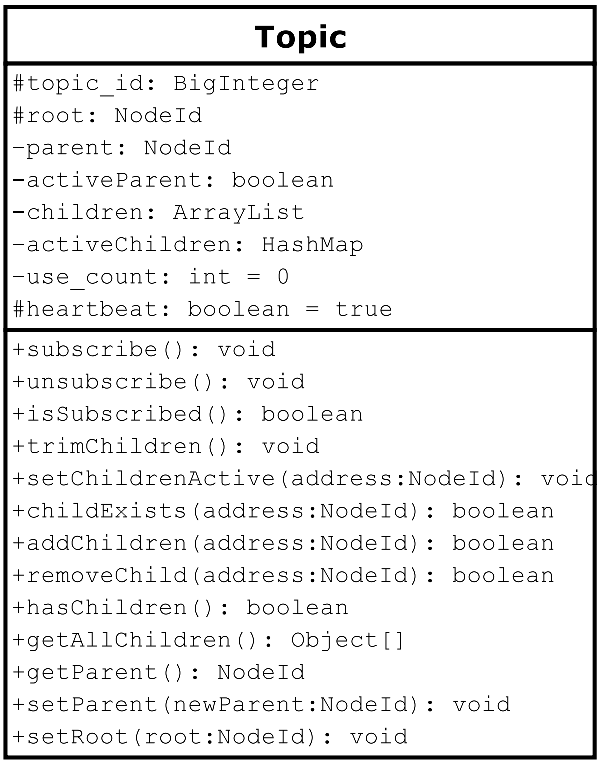
In the following I will describe how the reverse path forward protocol has been implemented.

3.4.2.1 Protocol procedure

Figure 3.9 visualizes the mechanisms of the multicast protocol. It is slighlty simplified for clarity, all messages from an application to the multicast layer are ment to be `PublishMsg` events, messages from the multicast layer to the application are `IncomingMsg` events.

First a CREATE message is sent to the multcast layer and routed through Bamboo to the node numerically closest to the topicID (without upcalls). Afterwards the application sends a SUBSCRIBE message. Again it is routed through Bamboo, but this time with upcalls enabled. A node inbetween gets the upcall and as it does not recognize the topic so far (in this case) it discards the SUBSCRIBE message, adds the source to its children for that group and tries to subscribe itself. Finally, the message hits the rendez-vous point. The source of the message is added to the children of the rendez-vous point.

Then the application publishes a new message to the group. If it would know the root's address it could use a `NodeMsg` message and send it directly. But in this case



```
Topic
#topic_id: BigInteger
#root: NodeId
-parent: NodeId
-activeParent: boolean
-children: ArrayList
-activeChildren: HashMap
-use_count: int = 0
#heartbeat: boolean = true
+subscribe(): void
+unsubscribe(): void
+isSubscribed(): boolean
+trimChildren(): void
+setChildrenActive(address:NodeId): voi
+childExists(address:NodeId): boolean
+addChildren(address:NodeId): boolean
+removeChild(address:NodeId): boolean
+hasChildren(): boolean
+getAllChildren(): Object[]
+getParent(): NodeId
+setParent(newParent:NodeId): void
+setRoot(root:NodeId): void
```

Figure 3.10: Class to store multicast group informations

– as the node does not know it – it is routed through Bamboo with no upcalls. The message reaches the root of the multicast tree and a `MCastMsg` with the address of the root is returned to the sender. Then the message is distributed to the children of the rendez-vous point and so on. When the node is locally subscribed to that group – which means that there is an application on that node that belongs to that group – the message is also sent to the local applications.

3.4.2.2 Multicast group management

Every node stores its information about the known – and therefore also subscribed – topics in a linked list. The needed informations about a topic is stored in a `topic`-object which you can see in figure 3.10. No node knows the whole multicast group but only the needed information like the address of the parent or the root and its children to this group. A node saves such an object if either it is root and called to create a new topic or if it gets a SUBSCRIBE message and has not known the topic before.

The topic class has fields to store the address of a parent, the address of the root of the multicast tree and an array for the addresses of all children. `activeParent` and `activeChildren` store information if there was a life-sign recently to detect failures, either of a child or a parent.

The `use_count` variable counts the number of locally subscribed applications. The boolean `heartbeat` indicates if HEARTBEAT messages should be sent to the children of this topic. This is an optimiziation switch, because normal PUBLISH messages act as heartbeats, too, and can replace explicit HEARTBEAT messages, especially when there are frequently published events for a group.

The function `trimChildren()` purges all children from which there was no lifesign recently from the children's array.

To enable a consistent group management, the following rules have to be obeyed per group:

- It is assumed that root-nodes have no parent, but store their own address in the `root` variable.
- Every time a message from a parent is received, its address is written with `setParent(address)`; this also sets the parent active.
- You must prevent to add your own node to the children's array, otherwise you will get an infinite sending loop.

3.4.2.3 Multicast group maintainance

There are two mechanisms in ComCore that maintain the multicast tree: A so called `ChildAlarm` and the `MCastRepairAlarm`.

The `ChildAlarm` makes sure that a heartbeat is sent to each children of each group known by the local node. Also acknowledgements are passed to a parent if there was none already within the last time periode. By default this alarm is initiated every ten seconds.

The `MCastRepairAlarm` – as the name suggests – tries to detect failures of children or parents and to repair the multicast tree afterwards. It is run every 30 seconds by default.

If a child fails, which means the node has not got an ACKOWLEDGMENT message from it since the last run, the child is deleted from the children table. This is done by the `trimChildren()` method of each topic object. If the child is marked as inactive, it is removed. Otherwise the active flag is set to false. Topics with an empty children table and no local subscription are deleted and the node unsubscribes from that topic.

The same happens for a parent of a topic. If it is set inactive, which means there has been no HEARTBEAT or PUBLISH message since the last run, a new SUBSCRIBE message is sent to find a new parent. Otherwise the active flag of the parent is set to false.

Furthermore, a rendez-vous point sends a DISCOVER message to look if there is a "better" root out there. This is important, because if a new node joins the Bamboo network and its node ID is numerically closer to a topic ID than every other live node, `BambooRouteInit` messages to the rendez-vous point of a certain topic will be routed to this new node. But this node does not know of that topic and, of course, has no children to publish this messages to. This mechanism should prevent such problems.

All of these group repair and maintainance mechanisms rely on the choice of good parameters for the regular alarms (`MCastRepairAlarm` and `ChildAlarm`) as they indicate, how fast a failure can be detected. Using long durations between two timeouts result in very low maintainance overhead but in slow failure detection and

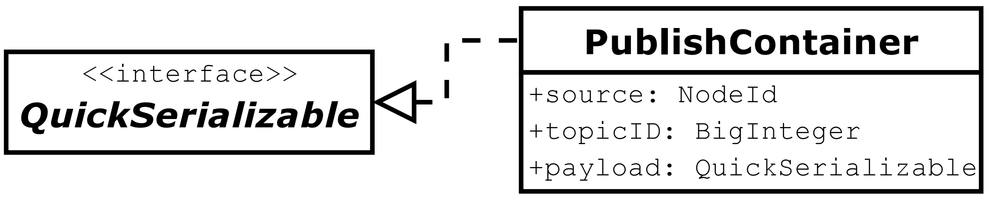


Figure 3.11: PublishContainer

repair. On the other hand, using very low timeouts result in fast detection but significant more overhead of maintainance messages. It was no goal of this project to optimize these parameters. Ten seconds were chosen for the `ChildAlarm` and thirty seconds for the `MCastRepairAlarm`. During the evaluation these parameters worked fine, but I did not look more closely on this – also as these durations are similar to those Pastry SCRIBE uses.

3.4.3 Special issues

The only time a node has the chance to cache the address of the actual root is, if it gets a NEWROOT message as an answer of a routed PUBLISH message. But `BambooRouteInit` messages do not support to add a sender address. But as `NetworkMessage`s do, I found it no good solution to extend each container with a source address, also as the application not necessarily knows the network address of the local node. Furthermore the usage of the multicast layer should be transparent and so ComCore does not need to know all the different containers that are published to the network – and so should not look inside them to gather information.

Figure 3.11 shows the payload I created to solve that problem. The necessary information for that event – topic ID and source address – are added and a payload can be inserted. The `PublishContainer` itself is then added as a payload to a `BambooRouteInit` message. This is transparent for the application using the multicast layer, because ComCore automatically wraps containers that must be routed and the root of the tree unwraps them again before publishing to the network.

As there is no access control implemented for the protocol so far, the CREATE message is not necessary. The protocol works fine without it at the moment, but in order to fulfill the API provided in the SCRIBE specification and to allow extending my implementation with some access control.

You must be very strict with your implementation in order to avoid severe failures. You have to avoid any cycles like adding the local node to its own children table which will cause an infinite sending loop. You have to make sure that there is only one root per group and avoid impossible states, like a root with a parent, at any time.

In the following chapter I will talk about the evaluation of the ComCore multicast layer and its interworking with the Bamboo peer-to-peer system.

4. Evaluation

In this chapter I will describe the evaluation of the multicast layer of my project. There exist measurements for Pastry Scribe and as I implemented its multicast protocol using Bamboo it is an interesting issue how my implementation will compare to the original one. As the protocol heavily depends on the routing mechanism of the underlying peer-to-peer system and its handling of locality this evaluation will also give conclusions about Bamboo and how its structure supports multicast applications.

4.1 PlanetLab

As a testbed I used the PlanetLab network. PlanetLab currently consists of over 350 nodes on more than 150 sites spanning about 20 countries. It is mostly hosted by research institutions. It provides a testbed for overlay networks and due to its structure a more realistic internet environment than testbeds in laboratories.

PlanetLab creates a unique environment in which to conduct experiments at Internet scale. All nodes are running on a common software package that includes a Linux-based operating system. A user can allocate slices of PlanetLab's network-wide hardware resources to an application and then perform experiments in a planetary-scale dimension, including real-world conditions of the network.

Real-world conditions also includes node failures and firewall problems which interfered my experiments several times. It is also not that easy to handle a large number of nodes. Already for 20 nodes it gets quite complex: You have to set up all nodes with the required software (in this case a Java Runtime Environment and Bamboo) and manage to start the experiments almost simultaneously. Furtunately there is *pssh*, a nice and useful Python script, that enables parallel *ssh* and *scp* execution and can be found on the PlanetLab website [Plan].

4.2 Preparation for experiments: EvaluationApp

The main goal of the experiments was to evaluate the implemented multicast layer. Therefore I needed an evaluation application to measure message delays and to

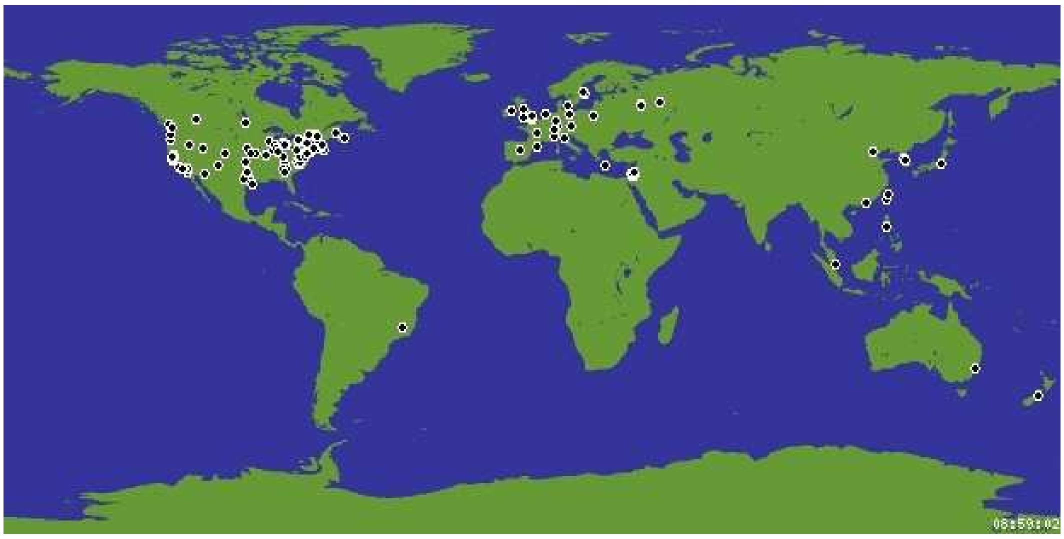


Figure 4.1: PlanetLab nodes

dump the created multicast tree. As I implemented the whole project in a layered structure, it was quite easy to replace the whiteboard with an evaluation application.

I also implemented an extended LineContainer with added sequence numbers and a timestamp. So the application measures the roundtrip time of a published message from its sending till its reception through the multicast tree. It then writes this value to a file.

Furthermore I added several triggers to affect the behaviour of the application, which can be set in the configuration file:

- **topic** sets the topic name.
- **listener** controls if packages should be sent by this instance or if it just should listen to a topic.
- **changeOffset** increases the time between regular actions (like topic changes and message publishing) in seconds.
- **topicChange** sets the probability for a topic change in terms (1 / topicChange). 0 means no topic changes.
- **duration** sets the length of the breaks between two publishs.
- **runningTime** sets the time the evaluation should run in minutes. The node shuts down after that time.

I also extended the ComCore class so that it dumps all its multicast group tables every 30 seconds to a file. So it is easy to reconstruct the used multicast tree.

4.3 Experiment: Tree-build evaluation

The goal of this experiment was to compare the implemented peer-to-peer multicast communication approach with an IP multicast implementation in terms of tree build-up.

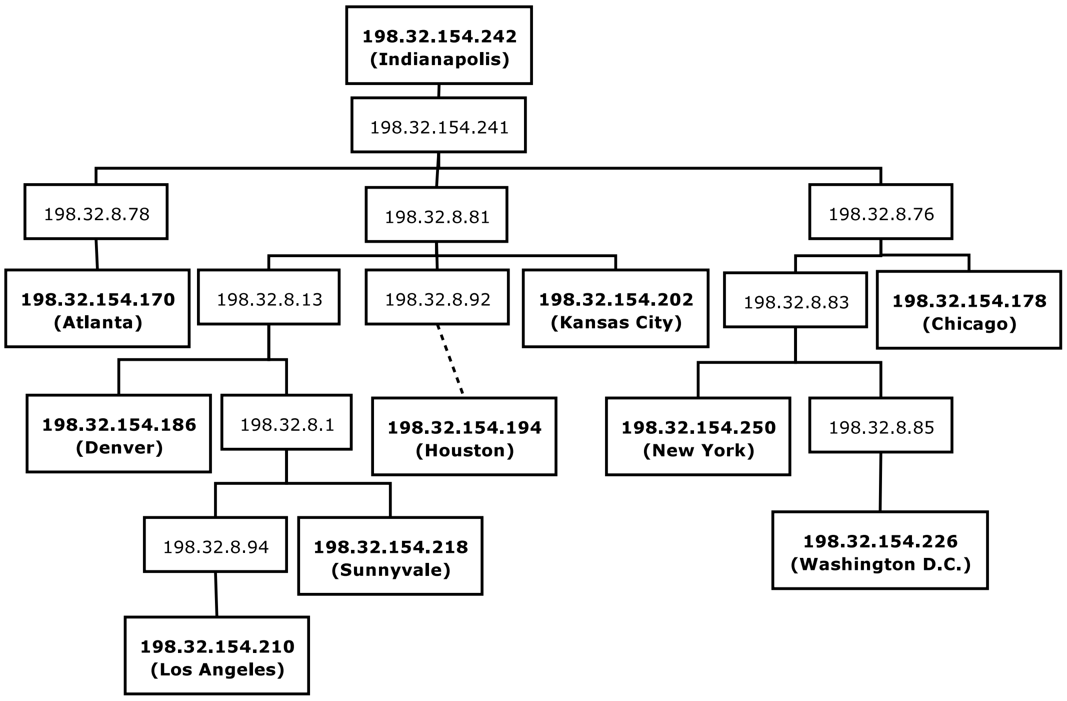


Figure 4.2: Internet2 IP multicast tree

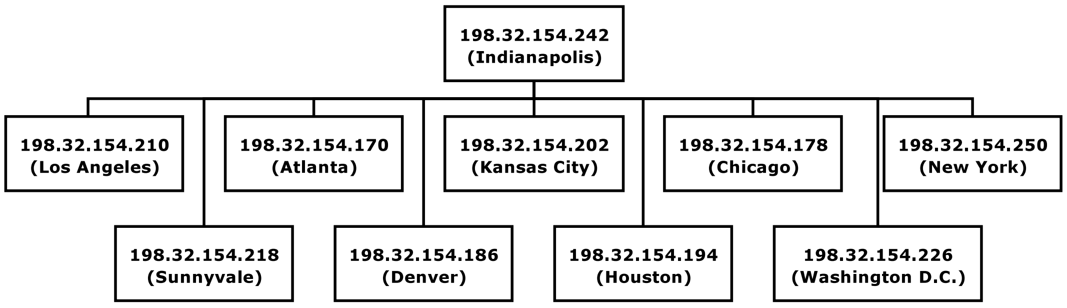


Figure 4.3: Internet2 ComCore multicast tree

Every node acted as an active sender to one common topic. Each node published a burst of 8 `LineContainers` to the group every 40 seconds. The experiment ran 60 minutes and the multicast group remained static during this time.

4.3.1 Comparison between IP and overlay multicast

The measurement of the IP multicast part is done by using `traceroute` to determine the routes from the rendez-vous point to each node. It is defined by the node chosen by Bamboo (numerically closest to the selected group ID).

As PlanetLab is very widespread over the planet I took for the multicast comparison experiment nodes that are assumed to use a quite common infrastructure – in this case 10 nodes from the Internet2 network located in the USA. This should make sure that a dense multicast group is formed.

The comparison of the both resulting multicast trees can be seen in figures 4.2 and 4.3. The ComCore multicast tree results in a root with 9 leaves, while the IP multicast tree offers some common links to aggregate flows. As ComCore only sees the overlay network formed by Bamboo, the result is as good as expected, as in the

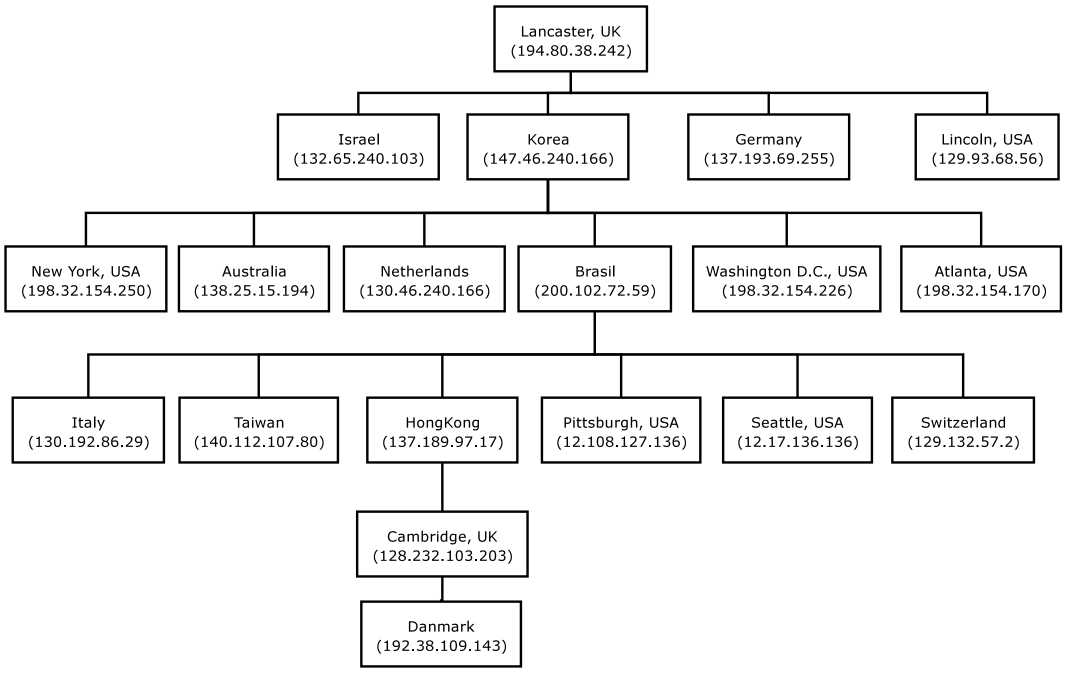

Lancaster, UK (194.80.38.242)
Israel (132.65.240.103)
Korea (147.46.240.166)
Germany (137.193.69.255)
Lincoln, USA (129.93.68.56)
New York, USA (198.32.154.250)
Australia (138.25.15.194)
Netherlands (130.46.240.166)
Brasil (200.102.72.59)
Washington D.C., USA (198.32.154.226)
Atlanta, USA (198.32.154.170)
Italy (130.192.86.29)
Taiwan (140.112.107.80)
HongKong (137.189.97.17)
Pittsburgh, USA (12.108.127.136)
Seattle, USA (12.17.136.136)
Switzerland (129.132.57.2)
Cambridge, UK (128.232.103.203)
Danmark (192.38.109.143)

Figure 4.4: ComCore multicast tree for 19 nodes

IP multicast tree the routers that aggregate and deaggregate the flows are not nodes of the Bamboo peer-to-peer network.

I also need to note that 9 nodes is a very small network and fits without a problem into the routing table and the leaf-set of each node, so that every node is reachable within an one hop range. This causes in every case a very flat multicast tree.

4.3.2 Conclusions from the experiment

In figure 4.4 you can see the resulting multicast tree of another experiment. This time there are 19 nodes arbitrary chosen and spread over the planet. As you can see the tree is higher this time. Taking a closer look at the tree reveals that the tree cannot be optimal. Routing a message from Lancaster (UK) via Korea, Brasil and Hong Kong to Cambridge (UK) does not seem to be very good (you can find more of these examples in the tree). As the implemented multicast protocol uses the Bamboo routing to build its tree, it depends heavily on its locality issues. As Bamboo uses PING messages to measure delays and choose nodes for a certain position in its routing table, many of these examples may be all right as the delay between those nodes is low.

But this does not explain that a message between Lancaster and Cambridge is routed once around the planet. In fact this is also an issue of the multicast protocol implementation, as there is no parent optimization. This means that a child keeps its parent until there is a problem with the parent node. So the shape of the multicast tree also depends on when a node joins the network.

Parent optimization on the other hand is not easy to achieve in terms of scalability and tree maintainance. It would also increase the complexity of the protocol a lot. Every child node would need to search periodically for a new node which increases the bandwith consumption. After discovering a “better” parent the local node would

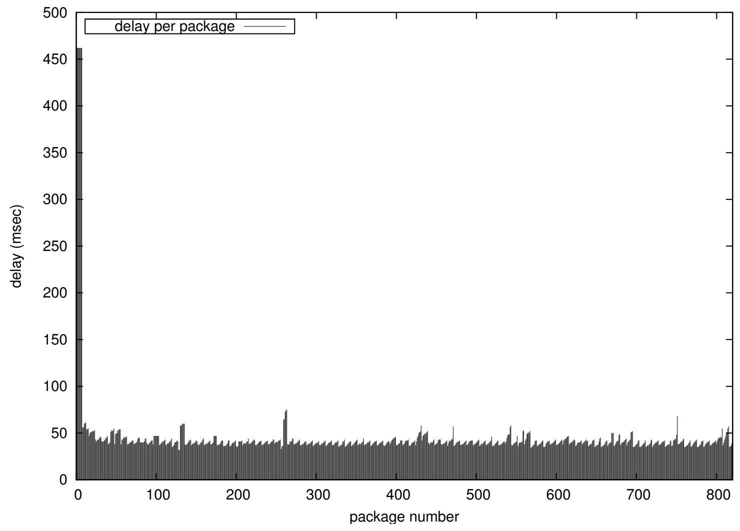


Figure 4.5: Delay for published packages (1 hop)

need to unsubscribe from its former parent and subscribe to the new one. In the worst case this can produce cycles for some short times while the node is subscribed to two nodes, or, in the other case, the node and all of its children would be disconnected from the multicast group for a while and lose packets.

Furthermore I discovered problems with the Java Virtual Machine if a node is flodded with packages, which can happen for a rendez-vous point of a large group with many published data at the same time. But this behaviour might only happen under extreme conditions, in this case 18 nodes publishing 16 packages each 8 seconds, which was due to a typo in the config file. This observation confirmed my decision to aggregate strokes and send them in bursts.

4.3.3 Bamboo delay

I finally took a look at the delay for sending messages to the rendez-vous point. In figure 4.5 you can see the delay of the node *planetlab1.wash.internet2.planet-lab.org* as an example. The peak at the beginning are the first 8 packages that are routed by Bamboo to the rendez-vous point and as expected this is a lot slower than direct sending, even – like in this case – when it reaches its goal within one hop. Afterwards the address of the root of the multicast tree is cached and the delay is a lot smaller. Compared to the traceroute roundtrip time for this path it is about double in length (that is the same for all leaf-nodes of the tree), at average a little more. This is called the *Relative Delay Penalty (RDP)* which is in the same dimension as evaluated for SCRIBE [CJKR+03].

In case of Bamboo, this delay is also related to the single-threaded architecture, so that every delay in the event handler also delays the processing of a package. In this case it should not affect the results, but it would be interesting to measure the delay for nodes with more than one application using Bamboo, so that more than one event handler is called when a message arrives.

5. Conclusions and outlook

This three months of project work in Cambridge increased my experience a lot in scientific research and work.

I had absolute freedom in my project work which I appreciated. So it was my task to choose a topic for my project and design it – there was only a framework provided by my supervisor. First I had to get into the topic of peer-to-peer systems and multicast networking. And as the underlying peer-to-peer system Bamboo lacks a lot of documentation I was forced to browse through the source code to learn how to use it which finally resulted in a tutorial that is now available at the Bamboo website [Disc04].

Several of my goals for that project are reached finally. I created a working multicast layer based on the Bamboo peer-to-peer system which can be used by different applications, not only by the whiteboard I implemented. The multicast protocol itself seems to work fine, also on a larger network like PlanetLab – even if I had not the time to deploy it on a large number (more than a hundred) of nodes – and the whiteboard is at least usable for simple tasks.

But especially the whiteboard lacks some features, for example that a new joining node gets the recent content of the canvas. It turned out that this is not easy to achieve, because of the strict seperation of multicast and application level I designed. As the multicast level works transparently for all using applications it cannot store any context for them (e.g. the current "official" content of the canvas at the root of the multicast tree). The new node has to find another node of the network that also runs a whiteboard application with the same topic and requests the content of the whiteboard from it. For this you can just go up the tree until you reach either a suitable node or a node with more than one child for that topic. At least in the leaves of the tree there are whiteboard applications situated that can be used. But it could happen that this node is also new in the network and requestes the actual content itself.

This problem would need some more consideration which I were not able to do because of lack of time. I already built a container class to transport the content of the canvas (called `ImageContainer`) for this purpose. While testing this, I found out that Bamboo is not fragmenting its messages (maximum message size is 16 KB), so this has to be done by the whiteboard itself.

Apart from that there are several ideas of how to go on. It would be pretty interesting to implement another multicast algorithm and compare it to the one used now. Another very interesting concept are Internet coordination systems which try to provide “real” locality to overlay networks. This would possibly improve the multicast trees a lot.

It would also be interesting to see other applications use my multicast layer. Actually the code will possibly be reused to implement video streaming on Bamboo. There was also the idea to use peer-to-peer multicast to interconnect IP multicast networks by tunneling non-multicast capable networks.

Bibliography

[CDHR02] M. Castro, P. Druschel, Y.C. Hu und A. Rowstron. Topology-aware routing in structured peer-to-peer overlay networks. http://research.microsoft.com/~antr/Pastry/, 2002.

[CJKR+03] Miguel Castro, Michael B. Jones, Anne-Marie Kermarrec, Antony Rowstron, Marvin Theimer, Helen Wang und Alec Wolman. An Evaluation of Scalable Application-level Multicast Built Using Peer-to-peer overlays. In *Infocom'03*, Apr 2003.

[DaMe78] Yogen K. Dalal und Robert Metcalfe. Reverse path forwarding of broadcast packets. *Communications of the ACM* 21(12), 1978, S. 1040–1048.

[Disc04] Marcel Dischinger. Bamboo: A Tutorial. http://www.bamboo-dht.org/tutorial.html, Apr 2004.

[KBCC+00] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells und B. Zhao. OceanStore: An Architecture for Global-Scale Persistent Storage. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000)*, Banff, Canada, Nov 2000.

[Plan] PlanetLab Website. http://www.planet-lab.org.

[RGRK03] Sean C. Rhea, Dennis Geels, Timothy Roscoe und John Kubiatowicz. Handling Churn in a DHT. Technischer Bericht UCB//CSD-03-1299, University of California, Berkely and Intel Research, Berkely, Dec 2003.

[Rhea04a] Sean Rhea. Bamboo: Programmer's guide. http://www.bamboo-dht.org/programmers-guide.html, 2004.

[Rhea04b] Sean Rhea. Bamboo Website. http://www.bamboo-dht.org, 2004.

[RKCD02] Antony Rowstron, Anne-Marie Kermarrec, Miguel Castro und Peter Druschel. Scribe: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communications (JSAC)* 20(8), Oct 2002.

[RoDr01] A. Rowstron und P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proceedings IFIP/ACM Middleware 2001, Heidelberg, Germany,* Nov 2001.

[Shir00] Clay Shirky. What is P2P... And what isn't? http://www.openp2p.com/pub/a/p2p/2000/11/24/shirky1-whatisp2p.html, Nov 2000.

[SMKK$^+$01] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek und Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM '01 Conference*, San Diego, California, August 2001.

[WeCB01] Matt Welsh, David Culler und Eric Brewer. SEDA: An architecture for well-conditioned, scalable internet services. In *Proceedings of the Eighteenth Symposium on Operating Systems Principles (SOSP-18)*, Oct 2001.

[ZHSR$^+$04] B.Y. Zhao, L. Huang, J. Stribling, S.C. Rhea, A.D. Joseph und J.D. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on selected areas in communications* 22(1), Jan 2004.

[ZYDM$^+$03] Nickolai Zeldovich, Alexander Yip, Frank Dabek, Robert T. Morris, David Mazières und M. Frans Kaashoek. Multiprocessor support for event-driven programs. In *In Proceedings of the 2003 USENIX Technical Conference*, San Antonio, Texas, Jun 2003. S. 239–252, section 2.