Postina: A Publish/Subscribe Middleware Designed for MMOGs

Master Thesis 2007-08

Dominik Zindel
http://postina.zindel.org

February 25, 2008

Content

MMOGs

- Massively multiplayer online game.
- Hundreds or thousands of simultaneous players.
- Example: World of Warcraft.

Motivation

- Scalability in MMOGs is a challenge.
- Client-server inflexible, inefficient, limits scalability.
- Goal: increase scalability and flexibility.

Requirements of MMOGs

- Scalability.
- Reliability.
- Consistency.
- Large group of receivers.
- Remote interest management.
- Handling of arrival/departure of clients.

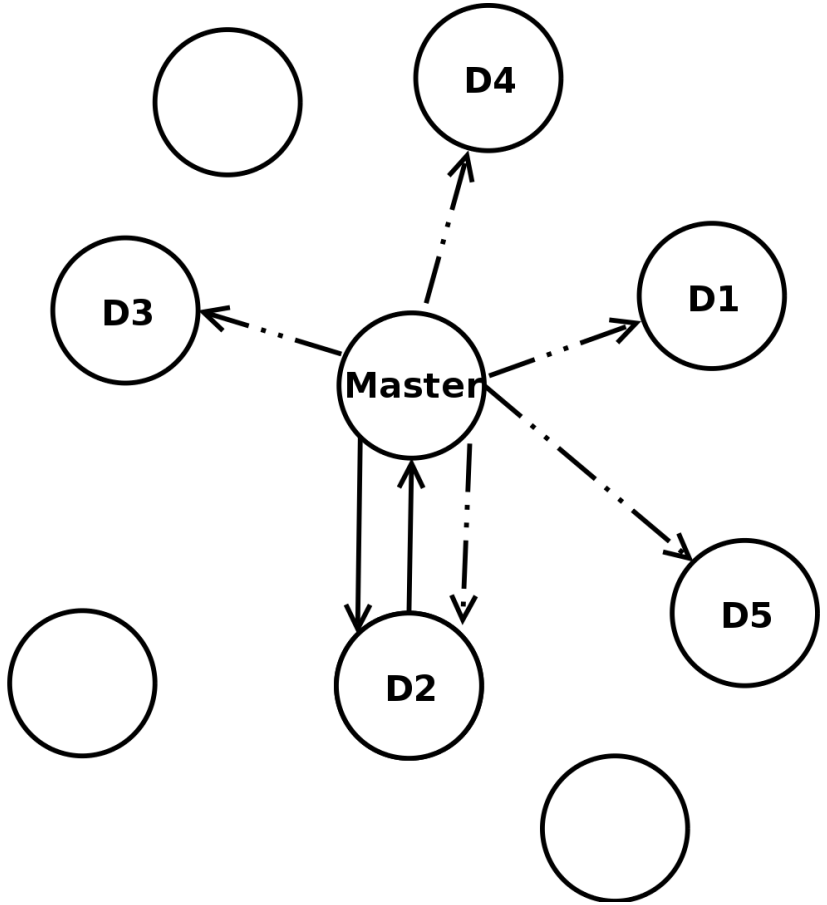
Duplication Space Approach

- Several players have to know about same items.
- Item is duplicated.
- One node contains original: master.
- Other nodes have a duplica/replica.
- State changes on master.
- Interest Management decides who is interested in change.
- Master informs duplicas.

Requirements II

- State updates: master $\leftrightarrow$ duplica.
- Send serialized data.
- Direct messaging.
- Multicast.

$\Rightarrow$ Publish/subscribe + direct messaging



Publish/Subscribe

- Asynchronous messaging paradigm.
- Publisher (sender) and subscriber (receiver) are separated.
- Subscribe: express interest.
- Publish: issue publication.
- Similar to observer pattern.

Three Actors

In a scalable publish/subscribe system, three types of actors are necessary:

Publishers or *producers* submit data as *publication*s or *notification*s.

Subscribers or *consumers* subscribe to publications, submit subscriptions.

Brokers or *event services* are neutral mediators between publishers and subscribers.

Publishers and subscribers are both *participants* or *clients*.

Overview of pub/sub

- Set of clients that asynchronously exchange notifications.
- Indirect communication, not point-to-point.
- Interaction between publisher and subscriber is mediated by a set of brokers.
- On receiving publication: broker determines subset of matching subscriptions.
- Subscribers are notified on matching publications by broker.

Selection Mechanisms

Channel-based selection With respect to a channel. All subscribers to this channel are notified. Inflexible. Example: all stocks of a market segment.

Topic-based selection Topics identified by keywords. Rather static. Every topic is seen as event service of its own. Example: `stock quote`.

Content-based selection Evaluate whole content of message.

Application Area of Publish/Subscribe

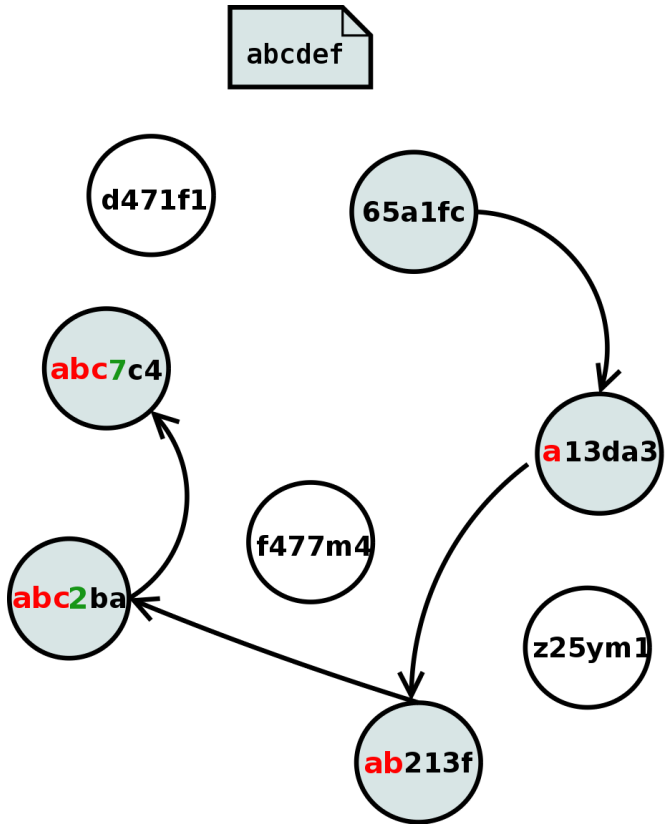
- Notify large number of subscribers.
- Scalability.
- Decoupling clients for flexibility.
- Asynchronous communication.

Description of Pastry

- Library for communication layer.
- Direct messaging.
- DHT: Distributed HashTable.
- P2P. Very scalable.
- Self-organizing.
- Each node: randomly chosen unique `nodeId`.
- Neighbours of node probably in different network.
- Each message: numeric key.

Routing Message

- Message routed to node with `nodeId` numerically closest to given key.
- *If destination in leaf set:* forward directly.
- *Else:* forward message to a node chosen based on prefix.
- Prefix shared must be longer than current.
- *If no such node:* forward to node sharing prefix of equal length but numerically closer.



Description of Scribe

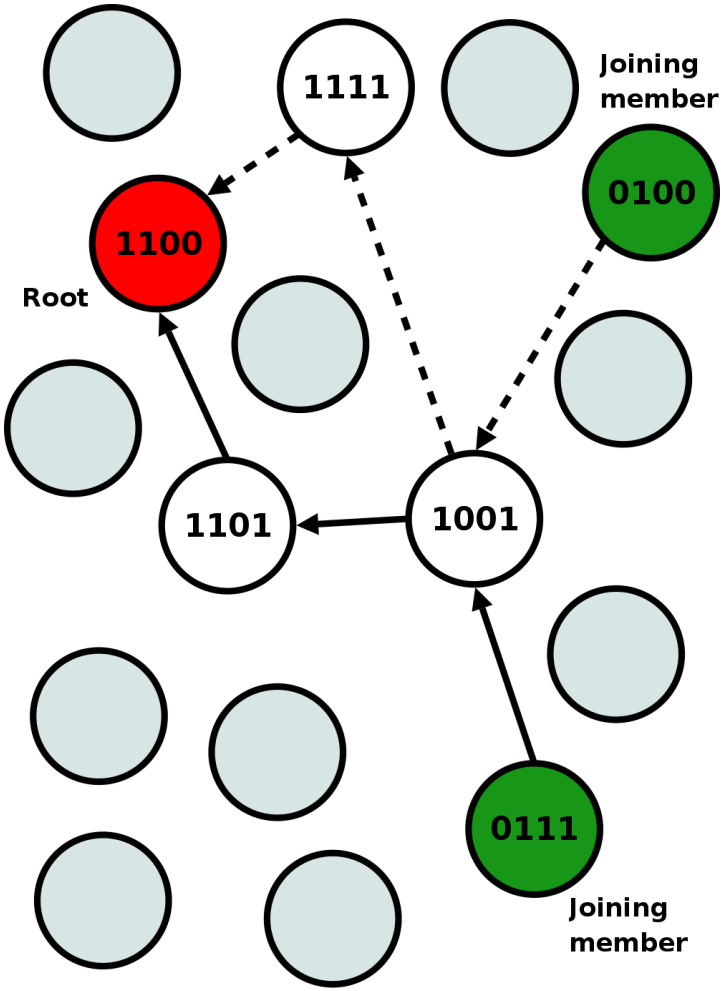
- Large-scale event notification infrastructure for topic-based publish-subscribe applications.
- Topic-based publish/subscribe system.
- Multicast.
- On top of Pastry.
- Fully decentralized.
- No reliability.

Topic Management

- `TopicId` for each topic.
- Equivalent to key in Pastry.
- Create topic: route special message with `topicId` as key.
- Routed to node *X* with `nodeId` closest to `topicId`.
- *X* is called rendez-vous node.
- *X* adds new topic to list of known topics.

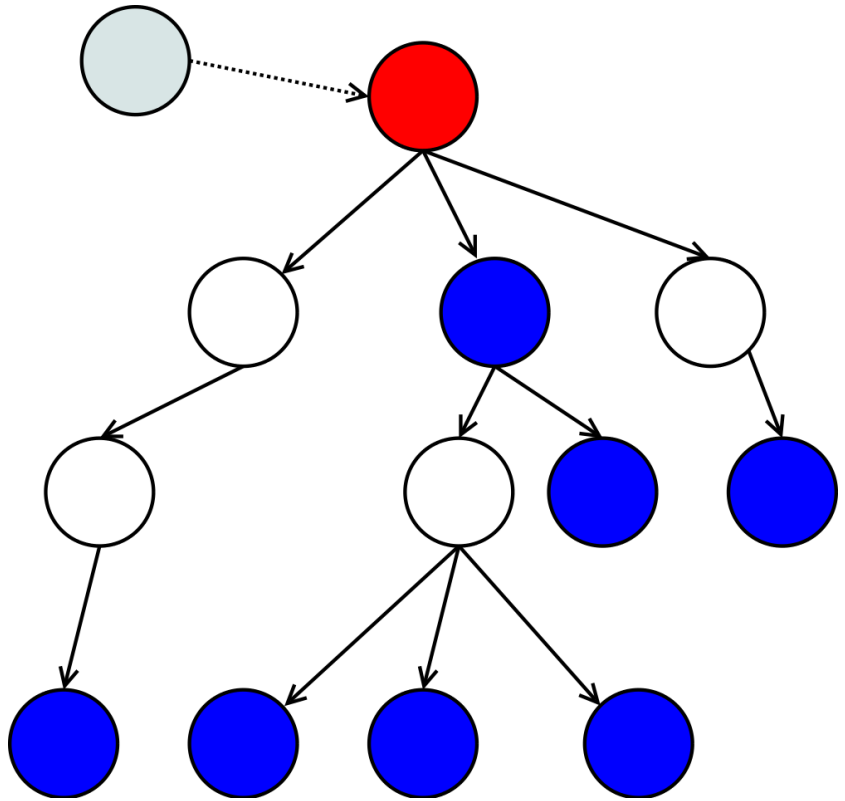
Subscribe

- Special message with `topicId` as key.
- Routed to rendez-vous node.
- At each node:
  if subscriber: add child.
  Else: add child, subscribe.
- Unsubscribe similar. Send unsubscribe if no children for topic left.



Routing

- Multicast tree.
- Nodes from subscriber to rendez-vous node.
- Each node: children table for each topic.
- Forward to children.
- To publish: send to root node (rendez-vous node).
- Distribute message load.



Description of Postina

- Romansh for “female mailcarrier”: pɔʃˈtɪnᵊ
- API/framework for network layer in MMOGs.
- Combines pub/sub and direct messaging.
- Special features.
- GNU LGPL.

Features

Publish/subscribe Subscribe, publish, connect.

Direct Messaging Send message directly to another client.

Remote Request Request another client to do something, e.g. subscribe to a topic.

Reliability Reliable direct messaging.

Dead Clients Detect and report dead clients.

Broadcast Send message to all clients in network.

Multiple Subscriptions Subscribe to multiple topics at once.

API

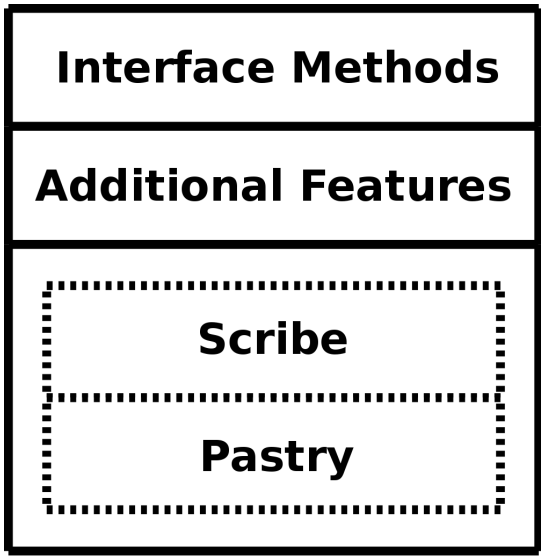
- broadcast(content:Serializable)
- connect(): PostinaID & disconnect()
- getTopic(topicName:String): PostinaTopic
- publish(topic:PostinaTopic,content:Serializable)
- subscribe by topic name, topic or collection<topic>
- subscribeOther
- send(destination:PostinaID,content:Serializable) & sendReliable
- Listeners
- Queue: getNextMessage()

Implementation: Why Scribe

- Publish/subscribe.
- Based on Pastry.
- Direct messages.
- Distributed environment.
- Efficient.

General Structure of Implementation

- 3 layers: implemented interface, additional features and Scribe.
- Implement additional features.
- Wrap Scribe/Pastry.

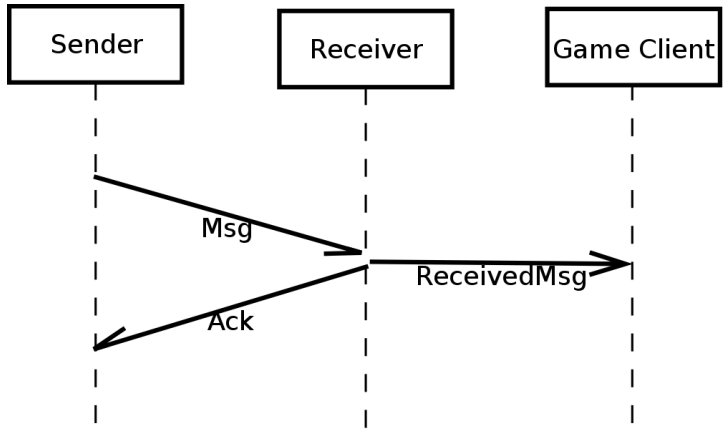


Reliability for Subscriptions

- Node failures, overload: subscriptions might be lost.
- Retry in case of failure.
- Keep track of outstanding subscriptions.
- Do not retry if unsubscribed in meantime.
- Transparent to user.

Reliability

- No reliability provided by Pastry.
- Added by Postina for direct messages.
- Using acknowledgements.
- Resends if acknowledgement does not arrive.
- Clients reject second delivery.
- Give up after maximum number of attempts.
- Order not guaranteed.



Dead Clients

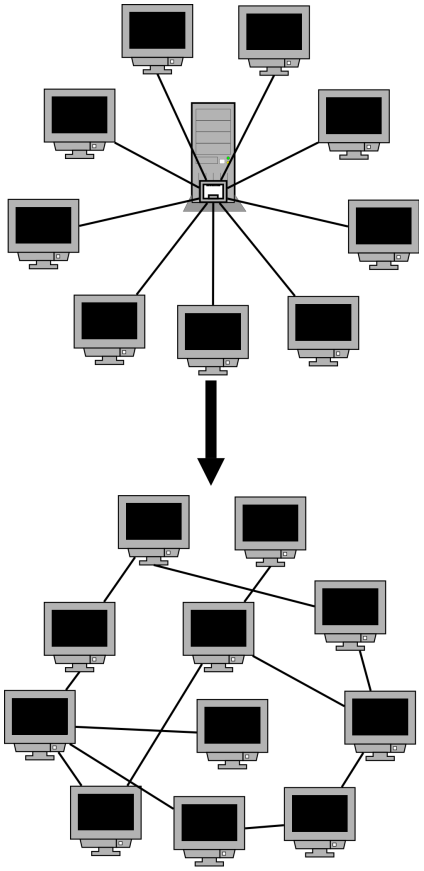
- Detect dead clients.
- Reliable messaging used.
- Message delivery fails definitely: suspect dead.
- Suspected dead several times: declared dead.
- Incoming message: pardoned.
- Report failure to listener & stop sending message.
- Done by each client.

Description of Mammoth

- Massively multiplayer game research framework.
- Role-playing game with simple actions.
- Duplication space approach.
- Master of all items on server.

Client vs. Server

- All nodes are equal in Postina.
- Mammoth uses a central server managing clients.
- Solution: server extends normal client.
- Server is node in network with random ID.
- Clients have to locate the server.



Locate Server

- Locate server at startup.
- Broadcast special request message.
- Server sends reply with its ID back.
- Client knows server, stores information.
- Retry if no answer.

Experimental Results

- Mammoth NPC running on SOCS machines.
- $>$ 300 connected players.
- 100 players in same interest range.
- Client-server only about 40 players.
- Fault tolerant.
- Postina efficient with many receivers of message.
- Few messages per node for a publication ($< 2^b = 16$).

Limitations

- P2P: firewalls and NAT prevent use. Ongoing work.
- Reliability limited to direct messaging.
- Limited quality of reliability.
- Direct messaging not very efficient.

Conclusion

- Combined publish/subscribe with direct messaging.
- Convenient interface.
- Scalable.
- Fulfilled requirements.
- Successfull integration into Mammoth.

- Interesting project in good environment.
- Nice experience at McGill and in Montréal!

Demo

Let's play!

Questions

http://postina.zindel.org