Colony II

SpiderCast: Distributed Membership and Messaging for HPC Platforms

Publish-Subscribe and DHT Services High Level Design

IBM Haifa Research Lab

Authors: Yoav Tock, Benjamin Mandler, Gennady Laventman

05/May//2010

TABLE OF CONTENTS

1 Introduction

This document presents the design of the DHT and Publish-Subscribe components of SpiderCast. This document is based on a document titled “An Architectural Overview and High Level Design”, dated January 2010 [TR1]. For completeness and ease of reading we bring a summary of the SpiderCast topology.

1.1 SpiderCast Topology Recap

The SpiderCast overlay is organized in a two tier hierarchical structure, comprised of “zones”, which are federated with a management layer (see Figure 1).

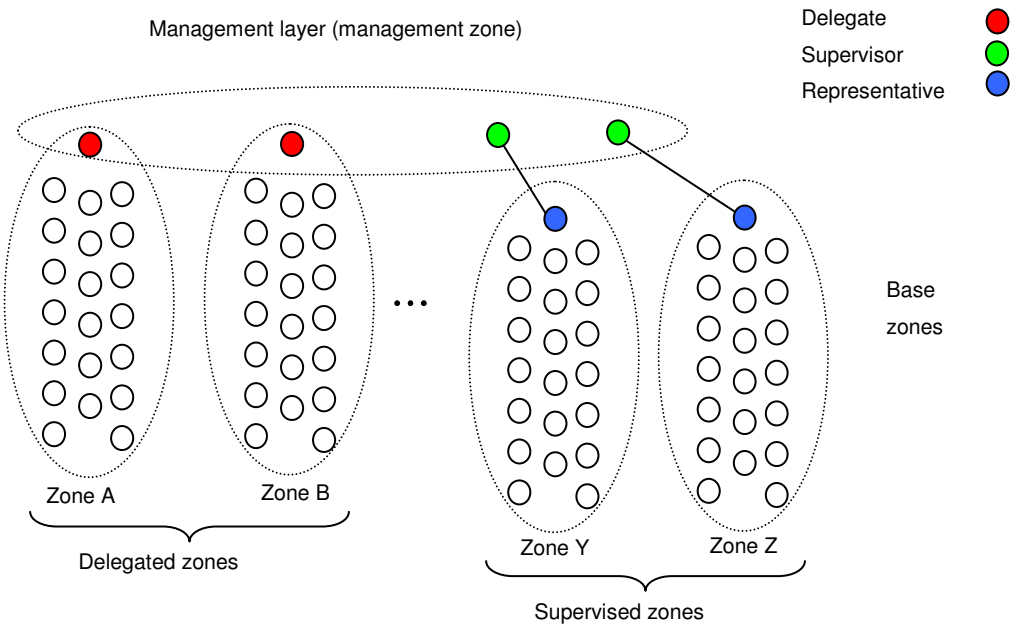

Figure 1 – Base zones federated by a management layer.

At the lower level of the hierarchy are base-zones. Base-zones are federated by a management layer, which forms a zone as well. The nodes which form the management layer can be either from within each base-zone (“delegates”), or nodes which are not part of a base-zone (“supervisors”). These two types of zones are called “delegated” or “supervised” zones, respectively. Each zone runs a distributed gossip-based membership algorithm. The nodes in

each zone have a full membership view of their zone members. A node can only be a member of at most a single base-zone. Nodes in the management-zone have full view of the management zone, and the full view of the base zone(s) they are responsible for. Thus a delegate in the management zone knows about all the nodes from the base-zone it belongs to and all the nodes from the management zone. A supervisor node has the membership of the management zone and the membership of the base zones it supervises by means of a protocol between the different entities. The protocol for connecting a base-zone with the management-zone can also tolerate the failure of supervisors, representatives and delegates. The topology of each zone contains a ring based on Virtual IDs (VIDs) which are derived from node names. In a sense, each base zone implements a structured overlay (see Figure 2 below, [TR1]).

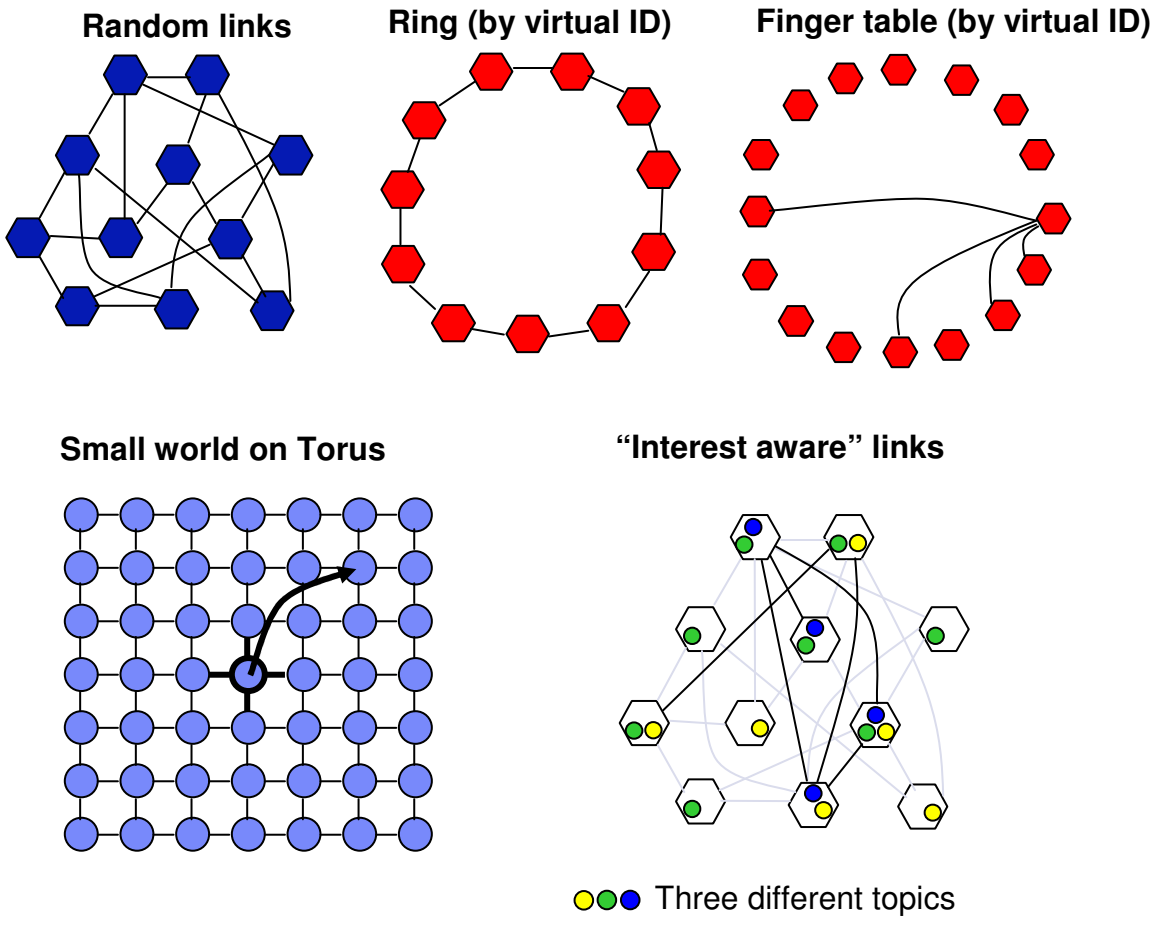

Figure 2 – Components of zone topology.

2 DHT – Distributed Hash Table

One of the features of our Spidercast design is the inclusion of a structured overlay per zone (an example of which is Chord [Stoica01]). A structured overlay naturally supports key based routing, which is instrumental in our pub/sub design. However, structured overlays were primarily conceived in the context of a distributed hash table (DHT). A DHT is a distributed structure that supports the storage and look-up of key-value pairs. The basic interface consists of [Dabek03]:

```
DHT.put( key, value )
value = DHT.get( key )
DHT.remove( key )
```

Every key-value pair is stored on some overlay node, according to the specific implementation of the DHT. Good implementations support efficient and scalable (O(Log(N))) insertion and look-up of key-value pairs. The DHT is a very useful distributed storage abstraction, and can be used, for example, to dedicate a chuck of a supercomputer compute nodes for an associative in memory storage (e.g. Amazon's Dynamo [DeCandia07]).

The well known implementations of DHTs operate in the following manner:

1) A node's virtual ID (VID) is a secure hash (e.g. SHA1) of its unique ID (say IP:port).
2) Nodes are organized in a ring according to their VIDs.
3) When a node wants to save a key-value pair the same hash function (e.g. SHA1) is invoked on the key, producing K-VID.
4) The storage node for the key is the node *N* who's virtual ID *N*-VID is smallest out of all participating nodes, but still holds N-VID >= K-VID. All ordering and relational operators are interpreted cyclically on the VID space.

Inserting and getting a key-value follow the same rule.

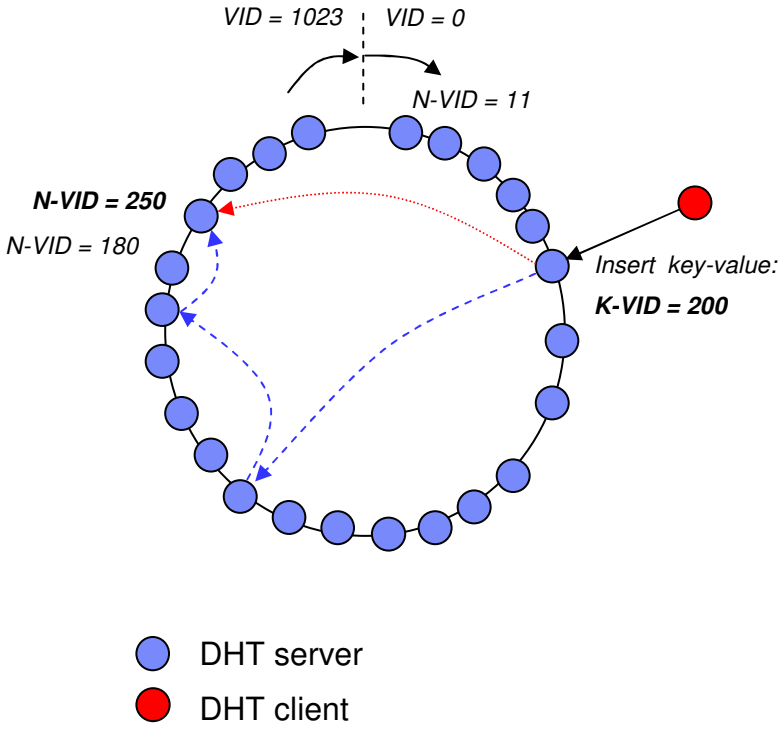

Figure 3 - DHT example on a 10 bit VID space: the storage node of key with VID=200 is node with VID=250

A DHT server is a node that participates in the distributed storage scheme. A DHT client binds to an arbitrary DHT server and accesses the distributed storage.

2.1 DHT in a Single Zone

DHT implementation in a single zone follows the design of well established systems. Every SpiderCast zone has full membership and implements a ring topology according to the node's VID (VID is computed from the node name, which is unique). As a result, a one-hop DHT can be implemented, meaning that the target storage node of every key-value pair is known to the source node. This type of DHT opens a direct connection to target node and performs the requested operation directly (e.g. Cassandra [Cass09]). Having full membership also eases the task of constructing a Chord-like finger table, which allows key based routing requests.

2.2 Multi-Zone DHT

The DHT service basically provides associative storage to the nodes that belong to the overlay. We differentiate between a DHT server and a DHT client. A server is a node that may actually store key-value pairs, and a client is a node that simply uses the DHT service. When a client is created a connection is formed to one or more servers that form an entry point to the DHT service, for that particular client.

We envision a system in which the number of DHT servers can be changed dynamically, according to the storage needs of the client nodes. The unit of allocation is a zone. A zone can be configured to provide DHT services, in which case all the nodes in the zone become servers. The DHT then grows and shrinks by adding and removing zones to the DHT. The DHT in each base-zone is called “base-zone-DHT”.

When a DHT-client is created it requests its delegate in the management-zone to provide it with an access point to the DHT, in the form of the ID of a DHT-server. That DHT-server then acts as a proxy for the client in all DHT requests.

The Upper Tier Ring

The delegate node (or nodes) of a base-zone-DHT in the management-zone mark themselves (using an attribute) as representing a DHT zone.

Each base-zone name is associated with a VID (secure-hash of the name). The delegates that represent DHT zones maintain a local data-structure of a ring from all the base-zone-DHTs, based on their zone-VIDs. Let us call this data-structure the upper-tier-ring (UTR). Building the UTR is a local operation because management nodes have full membership of the management-zone. Each entry in the UTR contains the name of the zone, and the IDs of the respective delegates.

The delegate of a base-zone-DHT propagates to its own base-zone the VID of the predecessor base-zone-DHT, as found in the UTR.

Lookup

Lookup of the target storage node ID that is responsible for a certain key works as follows.

1. A DHT-server which belongs to a base-zone is provided with a key.
2. The DHT-server calculates the key-VID (secure hash of the key value).
3. The DHT-server determines if the key-VID is served by the local zone. A key-VID is served by the local zone if: local-zone-VID >= key-VID > predecessor-zone-VID
4. If the key is served by the local zone, a normal DHT lookup is done (either one-hop or multi-hop, as in Cassandra and Chord, resp.).
5. If the key is served by a foreign zone

a. Send a request to the delegate to lookup the target-zone in the UTR, using the key-VID. Lookup in the UTR follows the same rules as lookup in a one-hop DHT.
b. The delegate (or the originator) then sends a request to the delegate of the target-zone, to lookup the target-node within the target-zone.

The server that originated the lookup may cache the key-VID, target-zone, and target-node triplet. This may speed future lookups of the same key. The caches are invalidated by delegates when they are notified of membership changes.

Replication

The data stored in a node may be replicated to guard against failures.

The data is first replicated to a predefined number of successors on the base-zone-DHT. This provides protection against failures of individual nodes. This scheme would be enough if node failures were independent. However, the block architecture of Blue Gene results in correlated failures. For example, a single faulty node may mean the node card (containing 32 nodes) may need to be shut down for repair. An entire zone may go offline because of a failed router, and so on. To combat this scenario we provide lateral replication – a node replicates its data to another node in the successor-zone according to the UTR. The target-node in the successor-zone that is the replica is the one that will take care of the origin-node VID, if it were a key inserted into its own base-zone-DHT.

Addressing uneven zone size

The scheme described above assigns the same number of keys to each zone, irrespective of its size. This works well in equal sized base-zones, but creates load imbalance if the zones have different sizes.

We therefore define the notion of a zone-slice. A global configuration parameter, *slice-size*, defines the number of nodes that compose a zone-slice (say 32 nodes). Each base-zone is virtually represented in the UTR by K=ceil(base-zone-size / slice-size) virtual nodes, by creating from the zone name K virtual zone names, using a suffix k=[1-K] . Note that the size of each zone is known to all nodes in the management zone, so again this is a local operation.

Thus, a zone “A” with 64 nodes is represented by two virtual slices named “A:1” and “A:2”; whereas a zone B with 1024 nodes is represented by 32 virtual slices “B:1” – “B:32”.

The lookup process described above works unchanged except that the UTR is now composed of slices, rather than zones. The target-zone lookup returns the zone that the target slice represents. This provides the appropriate weighing of zones and balances the key distribution.

Lateral replication works by returning (instead of the successor zone-slice) the first successor zone-slice that does not belong to the same base-zone as the source node.

2.3 Client API

The DHT is a distributed hash table implementation. It allows the client to insert key value pairs into the table, and look-up these pairs using the key. A DHT client is created from the Spidercast instance:

```
DHTClient Spidercast.createDHTClient (
      DHTClientConfig,
      DHTClientEventListener)
```

The client represents an access point to the table. The basic API has the following methods [Dabek03]:

Insert or replace a key-value pair:

```
Boolean     DHTClient.put(Key, Value)
```

Insert or replace a key-value pair, and retrieve the old value:

```
Value       DHTClient.putAndRetrieve(Key, Value)
```

Get a value using the key:

```
Value       DHTClient.get(Key)
```

Find if a key exists:

```
Boolean     DHTClient.containsKey(Key)
```

Remove a key-value pair:

```
Boolean     DHTClient.remove(Key)
```

A more advanced API allows write operations which are conditional on the value being written. This allows atomic read-modify-write operations on the map entries without the use of distributed locking.

Insert a key-value pair, if absent:

```
Boolean     DHTClient.putIfAbsent(Key, Value)
```

Replace a key-value pair, only if it exists:

```
Value       DHTClient.replace(Key, Value)
```

Replace a key's value with a **new** value only if the current value in the table equals the **old** value:

```
Boolean    DHTClient.replace(Key key, Value old, Value new)
```

Remove a key-value pair, only if the key is currently mapped to the given value:

```
Boolean    DHTClient.remove(Key, Value)
```

Note that a client cannot do operations that result in an access to the entire table, such as getting all the keys, all the values, getting the accurate size of the table, searching for a specific value, deleting the entire map, etc.

3 Publish Subscribe

Publish/subscribe (pub/sub) is a popular paradigm for supporting many-to-many communication in a distributed system (e.g. [Chockler01, Felber03]). Users interested in messages published on certain topics issue subscribe requests specifying their topics of interest. The pub/sub infrastructure then guarantees to distribute each newly published message to all the users that have expressed interest in the topic in question.

Many popular pub/sub solutions use either a centralized or a fully meshed underlying communication topology. Both approaches clearly do not scale in the case of a massively parallel computer. SpiderCast intends to build a decentralized infrastructure in which the processes are dynamically organized into an application level overlay network. The overlay created by SpiderCast is then used to route topic-specific events to all the processes interested in that topic.

SpiderCast's publish / subscribe capability refers to the topic-based flavor, in which the match between the publishers and the subscribers is established via their respective declaration of a particular topic name to be used. Conceptually, it is a broker-less fully decentralized design based on an Interest-Aware Membership service within and across zones. This paradigm is especially suited for distributed applications or systems that rely on short-lived usually large pieces of information distribution as a core element. Such a distributed system can be logically arranged into sub-groups via their subscription pattern to this service and enable an application group level multicast in which information is exchanged in an asynchronous fashion, and reaches only parties that have indicated a specific interest in such data. This paradigm is often associated with the Observer design pattern.

Since its inception the Publish / Subscribe model has been used in a plethora of different kinds of systems. The thin line connecting the different kind of uses lies with the asynchronous communication nature connecting different cooperating yet independent entities in a distributed environment.

An example use case is that of data dissemination. This use case is characterized by mostly having a single publisher pumping data into the system with potentially many subscribers interested in receiving the posted information. Generic data distribution, for replication, sharing, or other purposes, can be considered to belong to this family as well. For example, one component may publish the results of a computation which is needed as input for other components tasks. Parts of a work flow system may be built using this method. Such a scheme may be used as well to run cache coherency protocols, in which either invalidations or replacement updated data is communicated using the publish – subscribe mechanism.

3.1 Pub / Sub Operation Within a Zone

Within a zone the pub / sub service takes advantages of a couple of existing SpiderCast capabilities, namely full membership knowledge and the interest-aware membership service, which is a part of the general attribute service. Whenever a process subscribes to a topic, that information will be placed in its own attributes, and thus will be propagated to all other processes. Eventually all processes will know the identity of all the processes that have subscribed to all the topics. Gossip is used to track process membership, process attributes, and topic group membership. Topic group membership means tracking which processes are subscribed to every topic. Indirectly this service takes advantage of the routing capability which knows how to send a message from each member to each other member within a zone, using the structured topology. In order to optimize pub/sub routing, interest-aware links may be added. Those links are added according to the subscription patterns of the processes, in an attempt to improve the connectivity of a topic sub-graph [Chockler07]. Finally, an end-to-end reliability layer will be employed in order to ensure the delivery of all pub / sub messages to all receivers in the correct order. This layer is needed to support reliability over potentially multi-hop messages.

Topic objects are created via a call to a SpiderCast factory method. This factory method receives as input parameters a topic name and a configuration object. Topic name may be a general string, or a multicast group address like string (will be determined at a later stage). This method will generate a 128 bit Topic ID (TID), which will be used internally by subsequent SpiderCast operations. The transformation to a TID may be performed by a dedicated directory service, which may be distributed on each node or at a central location, or may simply be created by a local utility using a secure, collision-free, hash function. It is feasible for this information to be distributed across nodes in a management layer DHT, which in turn may serve as lookup service, both from topic name to its corresponding hash and vice versa.

Once a Topic object was created, both topic publishers and subscribers can be created.. As noted above, the attribute service will carry topic subscribers' information. Thus, there will be an attribute , named _interest for example, which will carry for each process the list of TIDs to which this process has subscribed. Routing within a zone may be optimized based on the amount of processes subscribed to a topic, as well as the distribution of these processes. For a topic with a small audience data may be sent in a point-to-point manner using either multi-hop routing over the overlay links, or by opening direct links from the publisher to the subscribers. For a topic with a large audience, broadcast may be employed [El-Ansary03]. Broadcast can be propagated over the overlay links efficiently and processes which are not interested in a specific piece of information may simply ignore it. An additional optimization may be put in place by which during the broadcast process complete sub-trees are pruned if it is identified that no process in that sub-tree is interested in the specific topic.

3.1.1 Reliability Modes

There are several reliability modes possible for pub / sub messages:

- Unreliable-Unordered - Best effort delivery. Messages are delivered to the application upon arrival, with no effort to order incoming messages or to reclaim lost messages, or to eliminate duplicate messages.
- Unreliable-Ordered – Messages will be delivered to the application in order and without duplicates, however messages may get lost.
- Ack-Based - Reliable ordered delivery based on explicit acknowledgements. Messages will be delivered to the application in order and without duplicates. The receiver will try to recover missing message by sending retransmission requests. The transmitter adjusts its history buffer based on information received from all receivers. Thus, the publisher needs to be aware of all receivers.
- Nack-Based - Reliable ordered delivery based on negative acknowledgements only. Messages will be delivered to the application in order and without duplicates. The receiver will try to recover missing message by sending retransmission requests. The transmitter adjusts its history buffer based on its history size. The transmitter does not expect to receive acks from the receivers; moreover, the transmitter does not need to be aware of the identity of the receivers. Thus, the transmission rate is determined by the transmitter, and slow receivers may fall behind and lose messages.

There is a transport protocol in place between the publishers and the subscribers, based on the reliability mode desired. In the following sub-sections we'll provide as an example some more details on a possible ack-based implementation. It will be determined in the future which mode(s) we will support.

3.1.2 Topic Publisher

The topic publisher determines the reliability mode from the configuration object. Upon the creation of a topic publisher, the new publisher instance obtains the current subscription list of the topic in question from the interest-aware membership service. The publisher initiates a handshake protocol with every subscriber in its list. Every subscriber that hasn't responded to the handshake after a predetermined amount of time is marked as failed, and an attempt is made to make the subscriber aware of this failure. Each publisher keeps track of the state of its relation with every subscriber (discovered, operational, …) as well as a message history buffer, from which it can re-transmit messages that haven't reached their destination. This buffer is trimmed once in a while based on the last message received by all subscribers.

Outgoing messages are numbered, and information regarding the oldest and newest messages available from the publisher is sent via periodic heartbeat messages to all subscribers. These

heartbeat messages may be piggybacked over data messages if transmitted during a certain time period.

In an ACK-based configuration, subscribers send an ACK to the publisher every configurable amount of time or amount of messages received. The ACK message will contain the last continuous message received as well as indication of missing messages. In response the publisher can trim its history buffer based on the collective response from all receivers and can re-transmit missing messages. When no ACK is received from a subscriber within a timeout period, that subscriber is marked as failed, and an attempt is made to make the subscriber aware of this failure.

When a publisher is terminated by the enclosing application it notifies all its subscribers of the upcoming termination. From that moment on, no additional messages can be sent via this publisher but it may remain alive for a while longer in order to service any incoming retransmission requests.

3.1.3 Topic Subscriber

When a subscriber is created by the application it will be inserted into the interest-aware membership and the attribute service will propagate this information to all processes in the zone. Each publisher upon receiving an indication pointing to a new subscriber will initiate a handshake process with that subscriber.

Upon receiving a handshake the subscriber sends an appropriate response, namely whether it wants to accept or reject that publisher. While accepting the connection, the subscriber will notify the application that a new publisher has been detected, and will create the data structures it needs in order to pass messages up to the application in the correct order as well as maintain all the information it needs in order to send periodic ACK messages to the publishers.

Upon receiving a data message it will be placed in internal buffers and if the ordering requirements have been fulfilled it will be delivered to the application.

When the time comes, the subscriber will send an ACK message to each publisher.

When the subscriber is closed by the application, the internal interest will be updated accordingly and propagated by the attribute service. The subscriber will let all the publishers know that it has been closed.

3.2 Pub / Sub Operation Across Zones

Publish / Subscribe should support different entities spanning multiple zones. A Multi-zone pub / sub service poses several challenges due to the potential scale of the overall system, and the

need to support different kinds of usage patterns, which not always coincide with the underlying zones distribution.

The main challenges in this respect are group membership and message routing, which come to solve both main questions: first, how does a publisher knows which processes are interested in a piece of information, and second, how does a publisher makes the information available to the interested parties. Group membership, including the interest-aware component, refers to tracking which processes are interested in every given topic. Within a zone full membership knowledge is maintained, thus every process is exposed to the interests of al other members in the zone. This mechanism is clearly not scalable, and thus a different solution needs to be devised for multi-zone scenarios. In the same manner, efficient routing from every source to every target is well supported within a zone, due to the full membership view, but a different mechanism needs to be put in place to support multi-zone traffic.

1) Group Membership:

Group membership means tracking which processes are interested in every given topic. Within zones that is achieved by the gossip mechanism. Every process in a zone knows about the interest of every other process. At the management layer the information is more coarse-grain, and is organized differently.

1) One possibility is for a supervisor/delegate to present the interest that is the union of the interests in its zone. Let us call that the “zone-interest”. The representative then propagates the zone-interest using gossip to every other member of the management zone, just like in any other base zone. The advantage of this scheme is its simplicity. The disadvantage is that every process in the management zone is exposed to the full list of topics, which can be very large.

2) In another option every topic is assigned a “home” process in the management layer, using consistent hashing. That (home) process tracks which of the zones is “interested” in that topic. Assuming that the total number of topics in the system may be very large, but that every process is interested in a small subset of those, this scheme ensures that no single process is ever exposed to the list of all topics in the system.

2) Messages Routing:

Routing will take place over the overlay links. Within each zone routing can be achieved by using the structured-overlay topology combined with known techniques for pub/sub (e.g. Scribe, [Castro02]). Routing between zones can be achieved in several manners:

1) Using the management layer for actually routing messages. The management layer has the complete inter-zone subscription information and thus, the representative of a publisher in the management zone can grab the data to be transmitted, pass it over to all

representatives of zones which are subscribed to the topic in question, and have these representatives in turn propagate the information within their respective zones.

2) Using the management layer as route "trackers" that respond to "route queries" and provide a route for messages between zones.

3) A preferred combination of several of the options mentioned above in which every topic is assigned a "home process" in the management layer, which keeps track of the zones interested in the topic, as well as a "home process" within each interested zone. The management home process for a topic keeps track of the per-zone home processes, and makes this information available in return to all zone home processes. Thus, a DHT is used at the management layer in order to determine the node which is responsible to keep track of a given topic. Whenever the first process within a zone declares its interest in a specific topic, by creating the respective topic subscriber or publisher, that information will flow to the zone representative in the management layer. Using a per zone distribution scheme, the representative will designate a specific process as the zone's topic gateway. From that moment on, the chosen process is in charge of propagating out all publications from within the zone, as well as distributing within the zone all publications originating in other zones.
The zone representative will update the topic's home node in the management layer as to the zone's interest as well as the zone's home process for that topic. In turn it will extract the current information held by the home process, and thus will hold the identities of all other zone home processes for that topic.
The zone home process is responsible for dissemination to all other zone home processes every piece of information that was published for this topic, as well as disseminating within its zone every piece of information for this topic which was published in other zones. In this scheme all the zone home processes will build a group (zone) of their own, whose membership will be derived from the topic's management home process. This scheme should achieve scalable and efficient topic interest group membership as well as information routing throughout the entire system. Within this newly formed group efficient dissemination trees can be built and maintained. Thus, publications should reach every interested zone with low latency, and the distribution within the zone will be made efficient as well due to the full membership knowledge. Both kinds of home processes will be determined in a manner by which it's trivial to locate its replacement in case of a process failure (should simply be the following process on the ring), and the information may be replicated accordingly.

3.3 API

The SpiderCast factory instance is used to create topics:

```
Topic SpidercastFactory.createTopic(
        TopicName,
        TopicConfig)
```

The **TopicName** is an identification of the topic in question. This is the property that needs to be agreed upon between publishers and subscribers. **TopicConfig** is a map-like configuration object, which encapsulates all the **Topic** parameters.

The SpiderCast instance is used to create subscribers and publishers:

```
TopicPublisher Spidercast.createTopicPublisher(
        TopicPublisherConfig,
        TopicPublisherEventListener,
        Topic)
```

The **TopicPublisherConfig** is a map-like configuration object, which encapsulates all the **TopicPublisher** parameters. The **TopicPublisherEventListener** object provides a callback for the delivery of life-cycle events that pertain to the specific topic publisher. The **Topic** object identifies the abstract channel used. The main usage of the topic publisher is to submit messages to the topic:

```
TopicPublisher.submitMessage( Message )
```

A subscriber is created in a similar fashion:

```
TopicSubscriber Spidercast.createTopicSubscriber(
        TopicSubscriberConfig,
        TopicSubscriberEventListener,
        MessageListener,
        Topic)
```

The configuration object, event listener, and topic carry the same meaning as in the publisher. The additional parameter **MessageListener** provides a callback method by which incoming messages are delivered to the application:

```
MessageListener.onMessage( Message )
```

Additional methods on the **TopicPublisher** and **TopicSubscriber** are:

- **`close()`**
- **`isOpen()`**
- **`getTopicName()`**

References

[TR1] Yoav Tock, Benjamin Mandler, “SpiderCast: Distributed Membership and Messaging for HPC Platforms: An Architectural Overview and High Level Design”. Colony-II technical report, January 2010.

[Dabek03] Frank Dabek, Ben Zhao, Peter Druschel, John Kubiatowicz, Ion Stoica, “Towards a Common API for Structured Peer-to-Peer Overlays”. IPTPS 2003.

[Decandia07] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: amazon's highly available key-value store," in *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, vol. 41, no. 6. New York, NY, USA: ACM, 2007, pp. 205-220.

[Stoica01] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," *SIGCOMM Comput. Commun. Rev.*, vol. 31, no. 4, pp. 149-160, October 2001.

[Cass09] Avinash Lakshman, Prashant Malik. “Cassandra - A Decentralized Structured Storage System”. *LADIS 2009*.

[Chockler01] G. Chockler, I. Keidar, and R. Vitenberg, "Group communication specifications: a comprehensive study," *ACM Computing Surveys*, vol. 33, no. 4, pp. 427-469, 2001.

[Felber03] P. Th, P. A. Felber, R. Guerraoui, and A. M. Kermarrec, "The many faces of publish/subscribe," *ACM Comput. Surv.*, vol. 35, no. 2, pp. 114-131, June 2003.

[Chockler07] G. Chockler, R. Melamed, Y. Tock, and R. Vitenberg, "SpiderCast: a scalable interest-aware overlay for topic-based pub/sub communication," in *DEBS '07: Proceedings of the 2007 inaugural international conference on Distributed event-based systems.* New York, NY, USA: ACM, 2007, pp. 14-25.

[Castro02] M. Castro, P. Druschel, A. M. Kermarrec, and A. I. T. Rowstron, "Scribe: a large-scale and decentralized application-level multicast infrastructure," *Selected Areas in Communications, IEEE Journal on*, vol. 20, no. 8, pp. 1489-1499, 2002.

[El-Ansary03] Sameh El-Ansary, Luc Onana Alima, Per Brand and Seif Haridi, Efficient Broadcast in Structured P2P Networks, The 2nd International Workshop On Peer-To-Peer Systems (IPTPS'03), (Berkeley, CA, USA), February 2003