Grant Agreement No.: 610764
Instrument: Collaborative Project
Call Identifier: FP7-ICT-2013-10

PANACEA

Proactive Autonomic Management of Cloud Resources

D2.5: Design of a consistency service

Version: 1.0

Work package	WP 2
Task	Task 2.4
Due date	01/02/2016
Submission date	
Deliverable lead	IBM
Version	1.0
Authors	IBM
Reviewers	Olivier Brun, Michel Diaz, Erol Gelenbe

Abstract	PANACEA proposes innovative solutions for proactive autonomic management of cloud resources. In this deliverable we introduce Frappe, a consistency service based on the Paxos protocol, which can be used for replicating the state among multiple nodes, thus serving as a basis for distributed systems. The basic algorithm is augmented by advanced features, such as dynamic reconfiguration of the replication set, and a speculative execution mode in which the system can advance even while going through a reconfiguration phase.
Keywords	Replicated State Machine, Configuration Services, Dynamic reconfiguration. Proactive execution

Document Revision History

Version	Date	Description of change	List of contributor(s)
v0.1	20.01.2016	Initial Skeleton	Artem Barger (IBM)
v0.2	30.01.2016	Internal review	Benny Mandler (IBM)
V1.0	05.02.2016	Final Integration	IBM

Disclaimer

Copyright notice

*R: report, P: prototype, D: demonstrator, O: other

Project co-funded by the European Commission in the 7th Framework Programme (2007-2013)		
Nature of the deliverable:	R	
Dissemination Level		
PU	Public	**X**
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to bodies determined by the PANACEA project	
CO	Confidential to PANACEA project and Commission Services	

EXECUTIVE SUMMARY

Distributed systems usually require consistency and coordination services for their proper day-to-day activities, and in particular to be able to withstand the failure of individual components. In particular, cloud and distributed middleware need a core component to store their critical operational information configuration[1]. This kind of information is requires to be provided with high availability guarantees, consistency and resiliency in the presence of failures. The Frappe service, described in this document, is a cloud middleware that provides highly available and fault tolerant building blocks for distributed applications to solve complex coordination tasks. The distributed locking mechanism is crucial to the PANACEA operation as it minimizes the conflicts in executing control operations in a distributed environment. The Frappe replication technology that is from the same family as Zookeeper[2] and Chubby[1] was built especially for coordinating data-centric distributed systems. It provides replication services[3], leader election and distributed locking and should be used wherever critical information is maintained. In particular, this includes: Replication Service for high availability, and Partition Tolerant leader election and locking mechanism. Frappe is designed to be hosted in the cloud and is ready for its elastic nature.

Frappe is used by all core components that maintain distributed state information that is required for the continuous operation of the overlay network component. This service is used to persist essential initial configuration for overlay routers. More specifically the overlay management system along with its routers uses the Configuration and Coordination Service's replication service to keep configuration about network topology and operational parameters required providing serviceability and availability of these components. Frappe leader election and membership APIs facilitate avoidance of split-brain scenarios during system operation, as only a single master can be available to orchestrate the cloud operations at any point in time. The group events API mechanism allows the overlay network component to receive notifications of new nodes joining the overlay.

Frappe introduces new algorithms to enable a reconfigurable state machine implementation and to facilitate elasticity of distributed replicated servers which must be able to adjust their size adapting to current and changing load on the system and effectively utilizing available resources. Moreover it allows higher-level services to leverage speculative command execution to ensure uninterrupted progress during the reconfiguration periods as well as in situations where failures prevent the reconfiguration agreement from being reached in a timely fashion. The Frappe service follows a black box approach that assumes nothing about the execution environment except the existence of reliable communication channels. In this deliverable we describe the system design, architecture and deployment models, list the main system API's and show interactions with additional Panacea components.

TABLE OF CONTENTS

LIST OF FIGURES

ABBREVIATIONS

AS	Autonomic Services
ML	Machine Learning
RSM	Replicated State Machine
SaaS	Software as a Service
VM	Virtual Machine
WP	Work Package

1 INTRODUCTION

Liveness and availability are key properties for many distributed systems; the primary technique to achieve those properties is to replicate the state and functionality of distributed systems so that the service as a whole can continue to function even as some components might fail. Usually, several instances of a single server are executed on separate computational nodes of a distributed system, to ensure that server failures are independent. Replication of the application state between server instances is very difficult to perform. Replicated state machine[3]–[5], or RSM, is an important technology for maintaining integrity of distributed applications and services in failure-prone data center and cloud computing environments. A state machine is a deterministic computation that for a given input and state generates a new output and moves to the new state[6], [7]. Paxos[8]–[10] is a consensus protocol that results in an agreement on an order of inputs among a group of replicas, especially in the presence of failures. By using Paxos to serialize the inputs of a state machine, the state machine can be replicated by running a copy on each of a set of computers and persist the state in the order determined by Paxos. Configuration and Coordination Service, the system described in this deliverable, efficiently implements a paxos based consensus algorithm.

In modern distributed systems, the infrastructure needs to adapt to changing resource availability, load fluctuations, variable power consumption, and data locality constraints. In order to meet these requirements, RSM must support reconfiguration, i.e., dynamic changes to a replica set, or quorum system. It is essential to ensure that reconfiguration incurs minimum disruption to availability and performance, in order to enable building truly elastic services. The ability to perform reconfigurations in a non-disruptive fashion provides system designers with a powerful paradigm that can enable many optimizations. This includes proactive replacement of suspected or slow nodes at low cost, adapting to changing environment characteristics (e.g. network delay, or diurnal load fluctuations, and many others). Frappe introduces a framework for constructing reconfigurable state machines from collections of non-reconfigurable ones. We follow a black box approach that assumes nothing about the execution environment except the existence of reliable communication channels.

The main ideas underlying our framework are the following. Each newly proposed configuration is associated with its own instance of RSM, and all active RSM are executed concurrently with each other. Gluing together the totally ordered command sequences produced by each RSM creates the globally consistent trunk of commands. When switching from one RSM to another, the latter is chosen based on the outcome of the configuration agreement in the former. Our framework also relieves RSMs of state transfer responsibilities by ensuring that the latest trunk is transferred to the new configuration concurrently with the RSM execution. This way, each newly created RSM is completely independent from its predecessor, and in particular, can start executing from its initial state. We leverage this capability in our Paxos-based reconfigurable RSM implementation to supply each newly created Paxos instance with the identifier of a deterministically chosen leader thus eschewing the first phase of Paxos if the configured leader does not fail.

The modularity of our framework enables a range of additional features useful in practical settings. One such feature is supporting rolling software upgrades: i.e., the implementation of the deployed RSMs can be replaced with a newer one without stopping the system. Likewise, misbehaving or buggy RSMs can be restarted or replaced on-the-fly with the minimum impact on the system operation as per the recovery-oriented computing (ROC) [11] guidelines.

1.1 Replicated state machine

A state machine replication based approach involves replicating a state machine on multiple instances of a system. Each of the state machine replicas begins with the same initial state. When the system receives a client request each of the replicas of the system processes the request and based on the output generated, updates its individual state. In a normal scenario the resultant state of each of the

state machine replicas will be identical. In order to achieve agreement and manage state updates consensus algorithm is usually used. There are two major stages in the leader based consensus algorithms: the leader election and the agreement stage (see figures below). Paxos is of the broadly adopted and well-known version of consensus algorithm. Every state update is replicated to all the servers, using a Paxos based algorithm, as described below.

1.1.1 Leader Election Stage

At any given time only a single replica node is eligible to serialize incoming requests thus imposing a total messages order for the rest of nodes in the replica set. Such a node is called a leader. Therefore the first step in the consensus protocol is to pick the "leader". If more then one leader is available at any point in time, the protocol will fail until only one node continues to act as a "leader". Therefore "leader election" is a crucial part of leader based consensus algorithms. Leader election works as follows: at start up replica with a smallest ID broadcasts to all other replicas proposal, trying to become a "leader", later the majority of replicas has to accept the proposal by sending an acknowledgement back. Once the chosen leader becomes unavailable a new round of leader election is initiated.

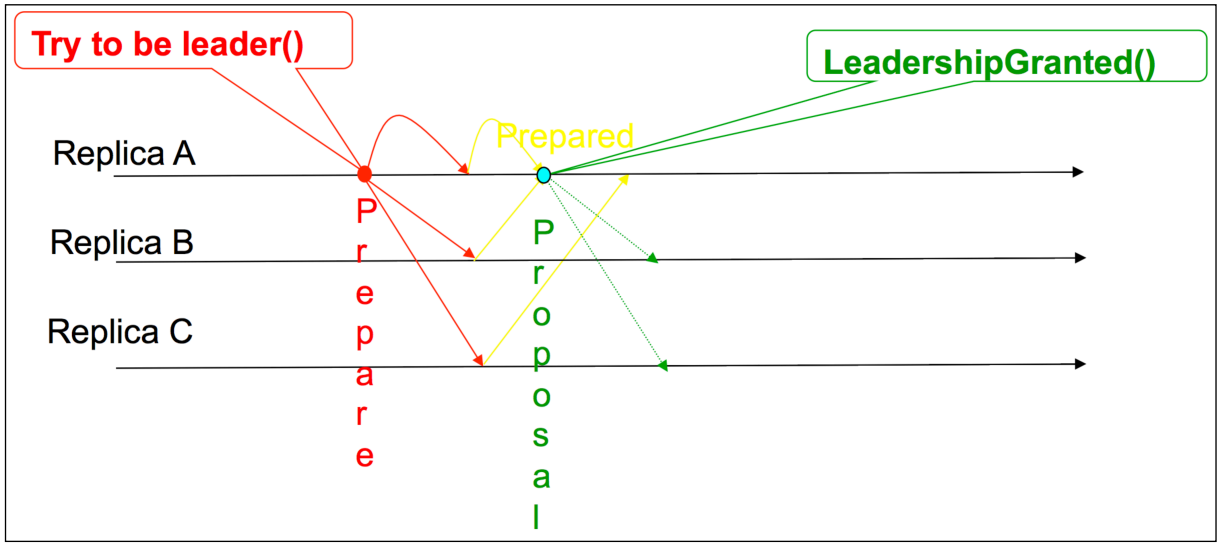

Figure 1: Leader election process

As we can see in the Figure 1, there are 3 replicas available: A, B and C. Replica A with smallest ID sends a leadership proposal to all other replicas B and C. After B and C receive a proposal they agree to accept A to be a leader, hence sending acknowledgements back. Because we have only 3 nodes in the replica set, the majority is any 2 of the currently available. Therefore A becomes a "leader", right after receiving an acknowledgement from replica B and its leadership is granted.

1.1.2 Paxos Replication

The Paxos replication algorithm is a quorum-based algorithm in which a write succeeds once a majority of the replicas accepts it. This style of replication can withstand F failures given 2F+1 replicas. Paxos can guarantee consistency under all conditions, including network partitions, and availability (liveness) under realistic conditions. Another advantage of Paxos is that it can tolerate slow replicas, and can gain progress even if some replicas are engaged in long running queries. With careful design, the read performance can scale linearly with the number of replicas.

Figure 2 shows replicated state machine architecture with front ends (FE), which propose commands (e.g. writes) at any order. The Paxos layer decides on a global order of commands that have been agreed upon by a quorum of the replicas. The replica back-ends (BE) see an identical stream of

commands. If the back-end is a database, the write command stream leave the database in an identical state in all replicas.

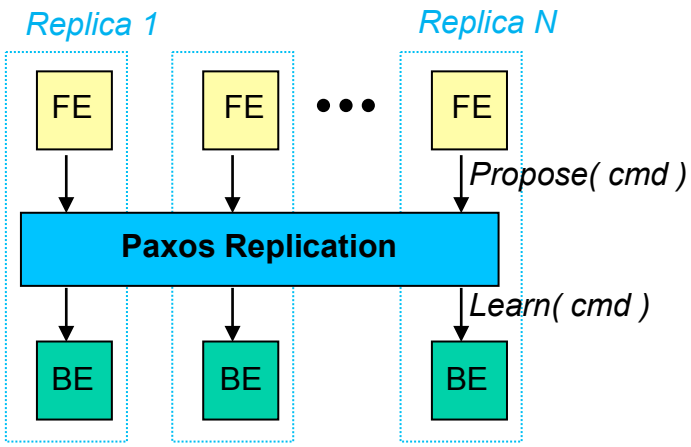

Figure 2 – Essentials of Paxos type replication

During network partitions, only the partition containing a majority of the replicas will continue to accept commands (writes). Thus no conflict resolution is needed after a partition is healed (at the cost of unavailability of writes in the minority partition). A mechanism for synchronizing back a node that is lagging behind or incorporating a fresh node into the system is described below.

This replication style fits the case where there is a need for highly consistent replication, with support for scalable read performance, and potentially support for long queries on some of the replicas.

The Paxos variant that we use supports 'reconfiguration' - the set of replicas can be dynamically changed without stopping the cluster - a tricky issue that is often glossed over in competing approaches and implementations. This feature facilitates dynamic cloud deployments and geospatially dispersed deployments. Our implementation also provides a framework for snapshot management and automatic state transfer after failure and network partition.

1.2 Frappe– a Paxos implementation

Our Paxos implementation provides member replicas with a mechanism implementing an abstraction of a totally ordered sequence of messages. Thus replicas can independently send messages via the Paxos framework and the framework will create a single global (i.e. shared by all replicas) totally ordered sequence of all these messages. This ordered sequence of messages is delivered to each replica by the framework.

The framework takes care of disseminating every message to all replicas, dealing with delivery of missed messages when a replica fails and then recovers, bringing new replica up to date with all needed messages, etc.

The core of the API of the Paxos framework:

- propose(msg) : Invoked by a replica to send a message to the cluster. This message will then be sequenced and eventually delivered to all the replicas.
- learn(msg): This is a callback in each replica that is invoked by the Paxos framework. The framework invokes this callback in order to deliver a message in the global total order to the replica. It is guaranteed that a message that is learnt at some replica will be learnt on all

replicas. Conceptually, the sequence of messages learnt by every replica is identical. Messages that have been sequenced are said to have been 'accepted' by the cluster.

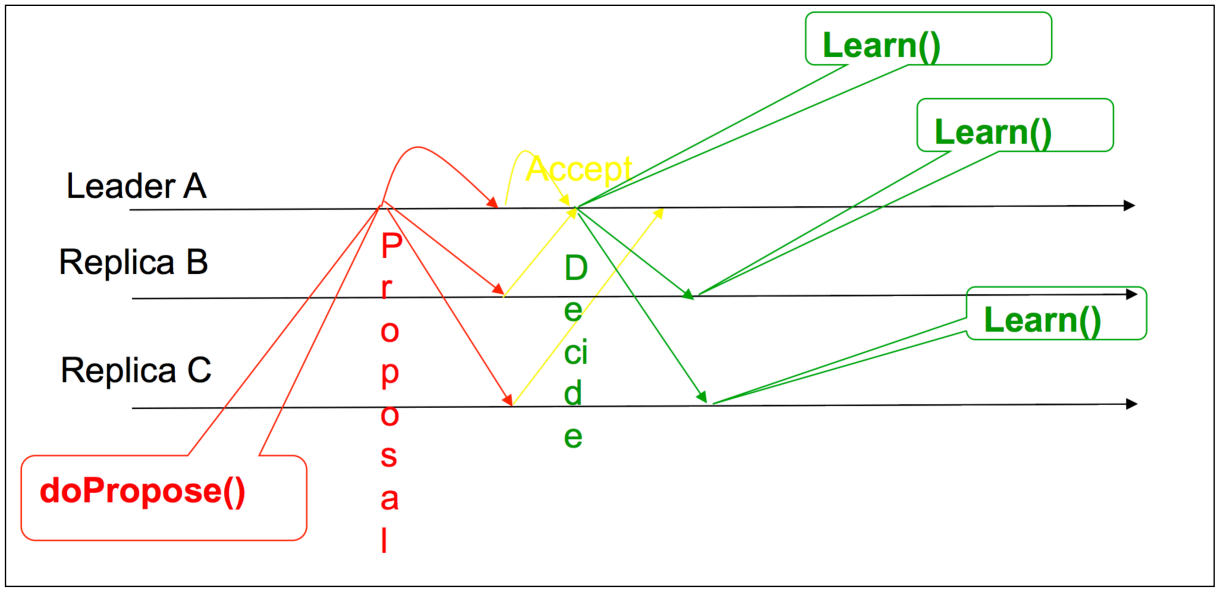

Figure 3: The Consensus algorithm

1.2.1 Failure Recovery and Elasticity.

If a replica fails or becomes inaccessible it will, once it becomes available again, be brought up to date with any state changes that occurred while it was inaccessible. There are two mechanisms for this and the system auto-selects the appropriate one at runtime. If the amount of write/update messages that occurred while the replica was unavailable is not too big, then the set of messages that it missed is sent. Once it processes these messages it is guaranteed to be up-to-date with the state of the other nodes in the cluster. If, however, the replica missed a large amount of messages, e.g. it was unavailable for a long period of time, a copy of the entire state is sent.

Frappe in Panacea supports dynamic enlargement and shrinking of the cluster (i.e. elasticity). The mechanism for bringing new replicas up-to-date with cluster state is similar to the one used for the failure recovery, in particular to the variant, which sends the entire state.

1.2.2 Speculative Command execution

Another important optimization made possible by the RSM independence is the ability to speculatively overlap their execution with the reconfiguration protocol, thus considerably reducing the command latency during reconfiguration periods. Specifically, each RSM is made available for accepting commands for the new configuration as soon as it is proposed, and without waiting for it to be agreed upon by the parent RSM. The proposals associated with the new configuration proceed to be ordered concurrently with the reconfiguration agreement, and are added to the trunk as soon as the configuration is agreed upon.

In the core of Frappe is a novel replicated state machine protocol, which employs speculative executions to ensure continuous operation during the reconfiguration periods as well as in situations where failures prevent the agreement on the next stable configuration from being reached in a timely fashion. Internally, our speculative state machine implementation is based on the reconfigurable Paxos approach i.e., configurations are treated as a part of the replicated state, and are being agreed upon in their own consensus instances. However, in contrast to reconfigurable Paxos, the command ordering can continue to execute normally even if the agreement on the configuration relative to which those

commands will be ordered is still in progress. The key observation is that the command ordering can be executed in an estimated configuration provided the validity of the speculative decisions can be verified once the next agreed configuration becomes available. To accomplish that, in Frappe, each replica maintains a branching command log as opposed to a linear log maintained by the standard replicated state machine implementations. Whenever a replica learns of (or proposes) a new speculative configuration, it creates a new log branch originating in the log location associated with the agreement instance created for that configuration.

Each branch then proceeds to execute its own independent sequence of the Paxos agreement instances. Whenever a replica learns the outcome of the agreement for a slot with one or more speculative branches, it stems all branches originating at that slot except for the one whose configuration is identical to the one that has been agreed (if exists). Speculatively agreed commands can be applied to the state either immediately (at the risk of the possible future rollback), or when their branch is validated against the agreed configuration. Since in a common case, estimated configurations would coincide with those being eventually decided, and the result of the reconfiguration agreement will be available by the time the first speculative command is agreed upon, reconfiguration will have a little impact on the overall command throughput. Furthermore, since incoming commands can continue to be ordered in an estimated configuration, the system availability will be unaffected even when underlying failures prevent the configuration agreement from completion. The global log consistency is guaranteed to be restored once the real configuration is learnt, and the speculative branches are verified against it.

1.3 Key Frappe Features

Novelties of the Frappe service could be summarized as follows:

- New generic framework for constructing reconfigurable RSM from non-reconfigurable ones, which is simple, modular, and efficient.
- New speculative approach to enable overlapping command ordering with the reconfiguration protocol thus reducing the reconfiguration latencies.
- Supports consistent replication of arbitrary stateful service via RSM.
- Supports pluggable replicated service logic and data model due to modular design paradigm.
- Mature framework, being an integral core component of WebSphere Liberty HA, now extracted to provide mature cloud ready service solution (SaaS), based on the cloud edge IBM technologies.

1.4 Frappe Services

1.4.1 Key-value configuration store

Service that implements a distributed, consistent key value store for shared configuration. Key-value service is layered on top of the Frappe Paxos implementation that is used to facilitate total order for all operations on configuration parameters thus providing strong consistency. This service is widely used by the Network Overlay component to maintain its initial topology information which includes destinations to which optimal paths have to be discovered.

1.4.2 Sessions

Service that provides functionality to create and manage distributed sessions. The Network Overlay routers required to initialize/create session during theirs life span in order to be able to leverage membership and leader election capabilities provided by the Frappe component and described below.

1.4.3 Groups and Leader election

Service that exposes an API to create a cluster of distributed nodes with coherent functional semantics by creating a virtual group. The Group service provides the ability for the nodes, which form a cluster to join or leave the group as well pub/sub functionality to register and receive consistent updates on events such as: node joined the group, nodes left the group, leader elected. Using the Frappe Paxos layer this service provides the ability to determine consistently and in a highly available fashion who is the current leader in the given group of distributed nodes, hence allowing the Network Overlay component to assure it operates with no split-brain between its components. Moreover it provides the ability for the overlay component to feed the system with correct and up to date information of overlay routers, which are currently alive within the system, and most importantly it allows to group routers together based on their roles in the overlay network.

2 ARCHITECTURAL RESPONSIBILITIES AND INTERACTIONS

The Figure below depicts the global architecture; most of the architecture components are described in great details in D2.4. Here we provide the description of interactions of Frappe with the entire PANACEA cloud components.

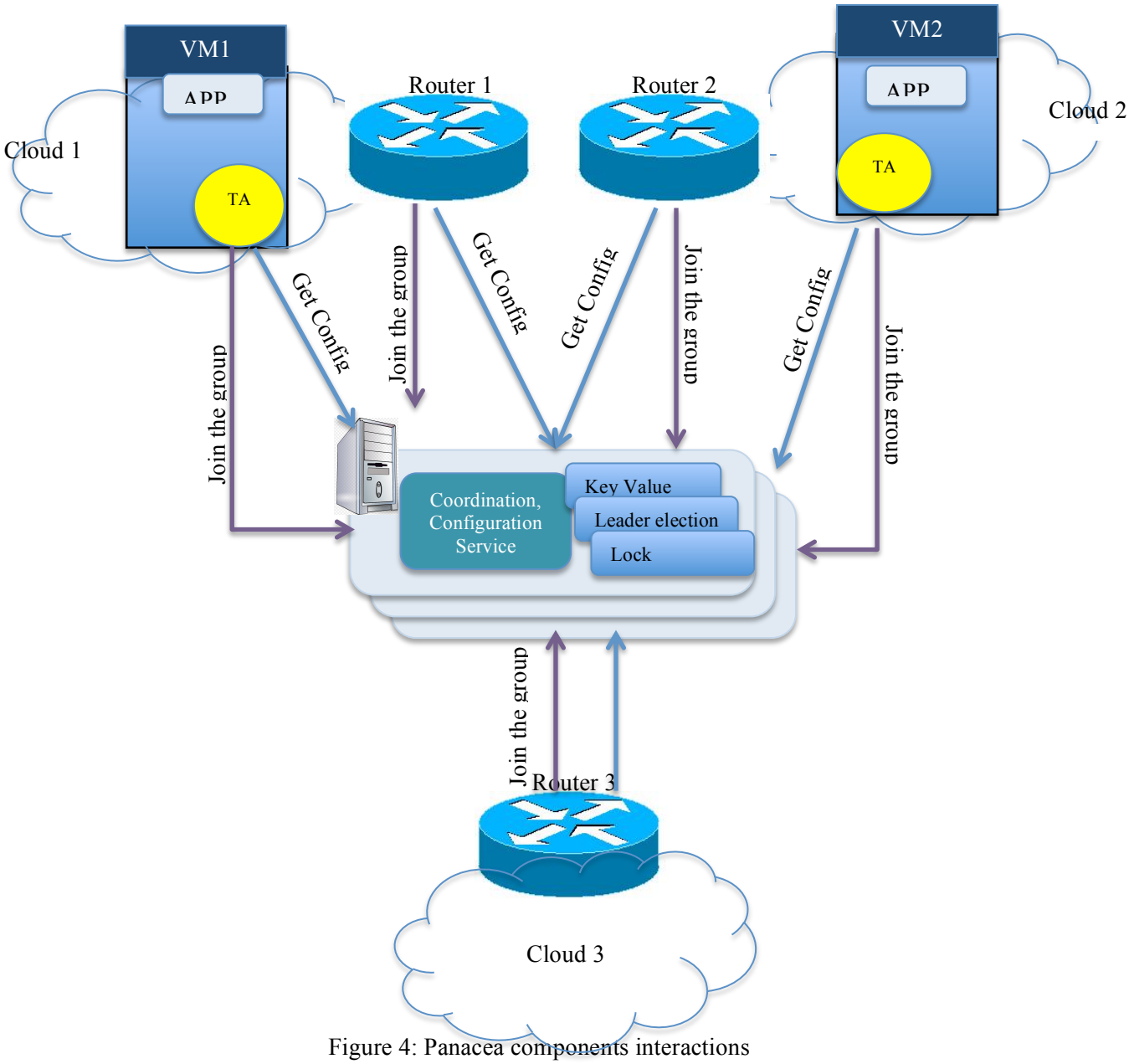

Figure 4: Panacea components interactions

2.1 Cloud Services Orchestration.

There are several routers and Transmission Agents (TA) deployed on the Cloud, whereas each TA is located in a different Virtual Machine (VM) and is responsible to intercept packets and forward them to the local proxy. Each router starts by joining the relevant group using Frappe Grouping API based on its role, in response it retrieves from the Frappe Configuration Key-Value service the initial information and configuration of the overlay topology. This information includes a list of the

destinations to which optimal routing paths have to be discovered. Moreover each router register to the event notification API to receive messages about new routers joining the overlay topology. Each of the network routers is required to use the membership capabilities of the coordination service to register itself to the set of nodes, which shares the same functionality. Using Frappe for overlay network guarantees failure tolerance of the network overlay components and avoidance of a single point of failure.

3 HIGH LEVEL DESIGN

The Frappe modular architecture embraces a pluggable design, allowing efficiently adding and implementing consistent and reliable services leveraging Paxos consensus protocol capabilities. The System architecture consists of the following key components:

- Frappe Core component – the implementation of a reconfigurable Paxos based consensus algorithm.
- Pluggable service – service that executes on top of the Paxos consensus algorithm to leverage globally serialized updates.

See below a high-level architecture service diagram (Figure 5). Frappe core is the actual implementation of a reconfigurable Paxos protocol; it provides an API for other services to execute agreement protocol on the set of distributed commands enforcing total order of command execution across the distributed setup of clients. In the Frappe solution we provide three pluggable services layered on top of Paxos, which are: Configuration Key-Value store, Session management and Membership with Leader Election.

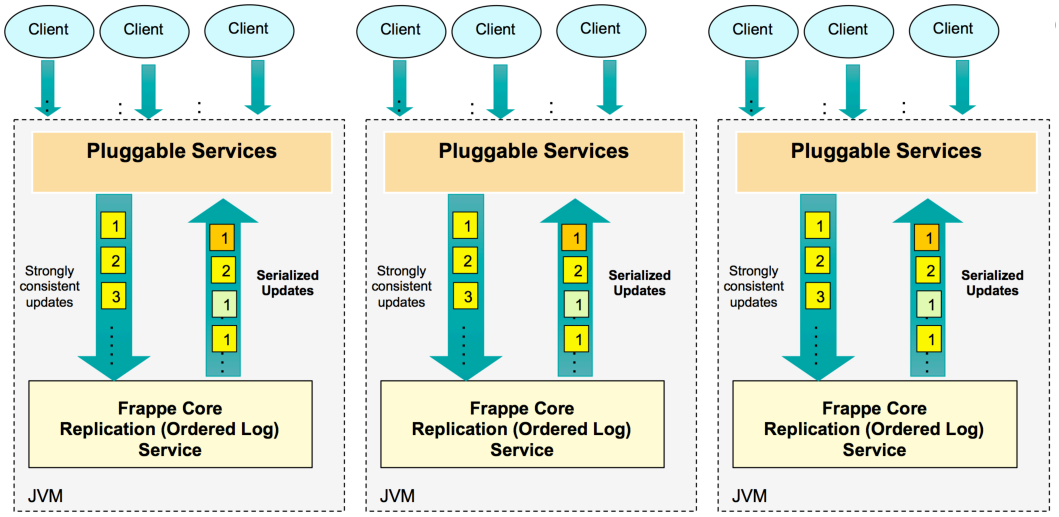

Figure 5: High level architecture

4 PERFORMANCE EVALUATION

We studied the performance of our implementation experimentally using a testbed comprised of 4 IBM HS22 blades equipped with Intel Xeon X5670 processors with 24 2.93GHz cores and 64GB RAM. Each machine was equipped with 1GB network card, and ran Red Hat Linux. A single replica was hosted on each machine.

The replicas were subjected to request streams generated in either a synchronous or an asynchronous fashion. In the synchronous mode, the requests were issued from multiple threads each of which was waiting for the response to the previously submitted request to arrive before submitting the new one. The synchronous mode allowed us to exercise high degree of control over the offered system load by varying the number of simultaneously executed synchronous threads. In contrast, in the asynchronous mode, each thread was submitting requests as they were generated without waiting for the prior requests completion. The asynchronous mode was used to drive the system to its maximum utilization.

In the first experiment, we studied the throughput and latency as a function of the offered system load in the absence of reconfiguration using a single configuration consisting of 3 replicas. The results (see Figure 6) show that our system is capable of achieving the sustained throughput of 12K request/second before the latency starts to rapidly grow.

In the next experiment, we studied the impact of our speculative reconfiguration mechanism on the command throughput. For that, the system was subjected to a series of reconfiguration requests submitted at varied frequency. The normal commands were submitted using 10 synchronous threads, which corresponds to the maximum sustained system load of 12K requests/second (see Figure 6). The measurements are depicted (see Figure 7) as absolute command throughput in the presence of reconfiguration, and percentage of degradation relative to the maximum sustained throughput (12K). The results show that the throughput in the presence of reconfigurations is almost identical to that achieved in the absence thereof, reaching maximum 20% of degradation when the system is reconfigured 20 times/second.

In the last experiment, we studied the degree of improvement produced by speculation in terms of the latencies of the commands submitted in close proximity to the reconfiguration requests. The results (see Figure 8) indicate that with speculation, the latency is unaffected by reconfiguration staying closely to that of the normal mode (i.e., in the absence of reconfiguration) throughout the entire reconfiguration period. In contrast, without speculation, the latency increases sharply for the first command, and then keeps decreasing slowly until reaching the normal mode value in the proximity of the 100th command.

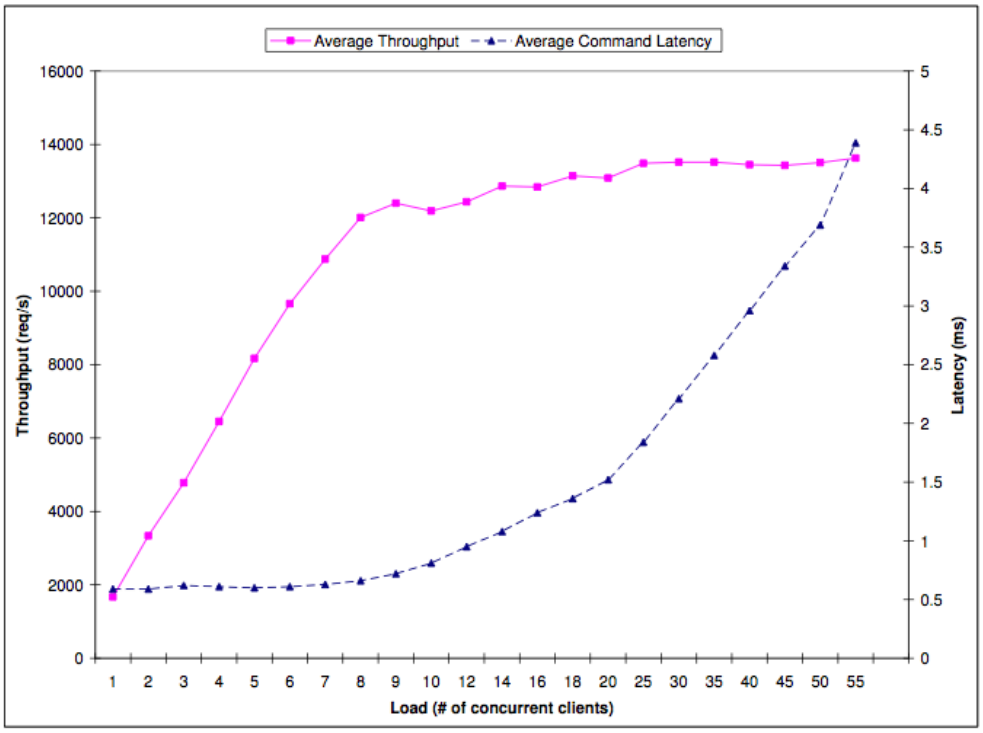

Figure 6: Throughput and Latency in the Absence of Reconfiguration

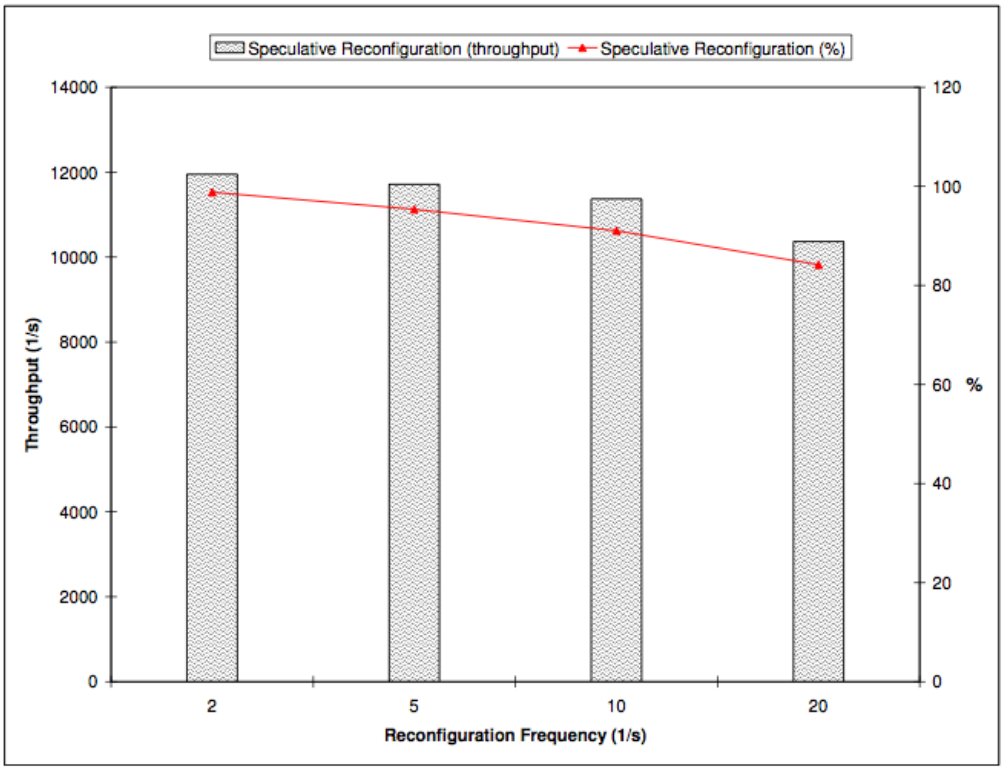

Figure 7: Throughput with Varied Reconfiguration Rate

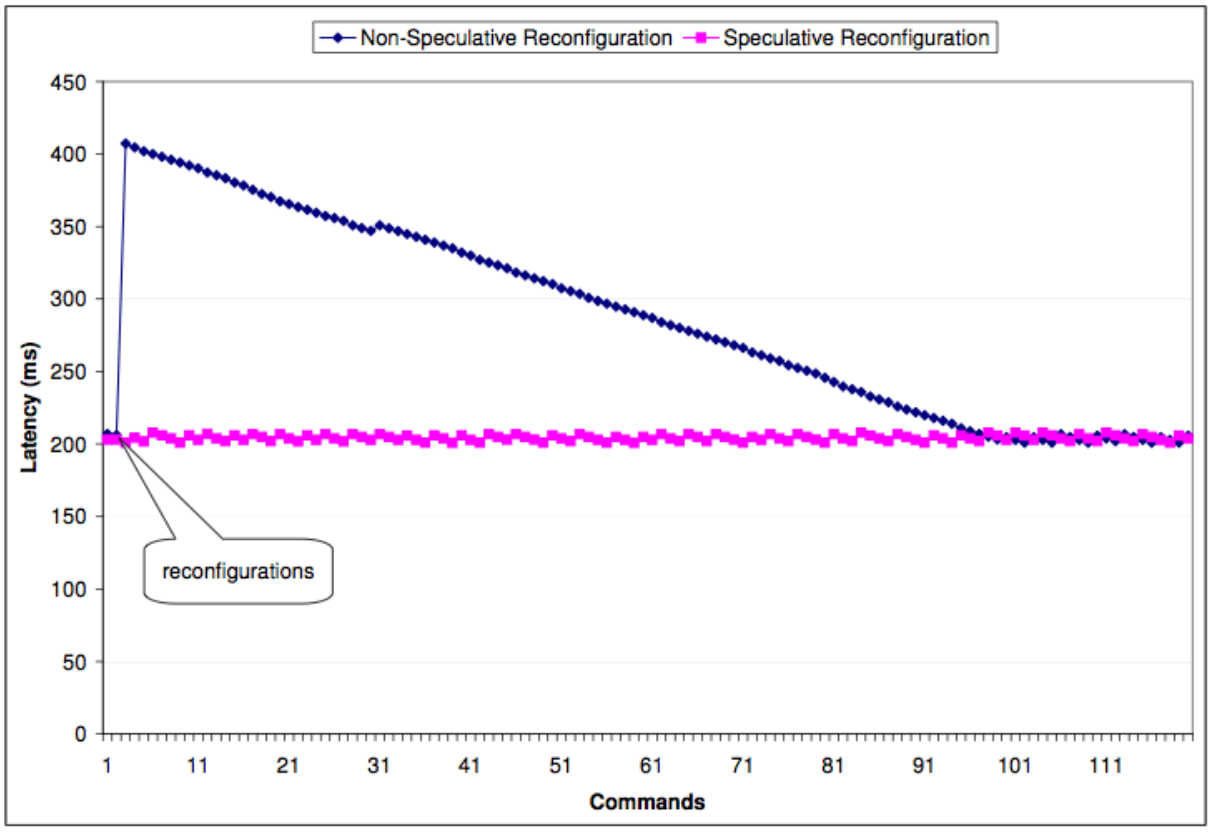

Figure 8: Command Latency in the Vicinity of Reconfiguration.

5 EXTERNAL API

5.1 Configuration repository

Service API's, which provides consistent, multitenant key-value data store capabilities. Main purpose is to provide configuration storage for distributed service or to facilitate persistence of service metadata in a reliable manner.

It has an endpoint: **/{tenantId}/v1/keys/{key:.*}**, where:

tenantId – is the tenant identifier
key – is the configuration key parameter

The HTTP **GET**, **PUT** and **DELETE** methods are all supported.

5.1.1 GET Method

When using HTTP GET, Frappe will respond with the value of the specified configuration key. If the "**?recursive**" query parameter is provided it will return the entire sub-tree of the given configuration key with its associated values.

Response will look like:

```
{"node":
      {"children": {"empty" : true },
       "value"   : "Hello World",
       "key"     : "/greeting"
      },
 "action": "getNode"
}
```

HTTP response codes:

Code	Description
200	Node found
404	Nodes does not exist

5.1.2 PUT Method

With the HTTP PUT method, Frappe expects to receive a request body with a value which corresponds to the key. If the "previousValue" parameter provided then it acts with compareAndSet semantics, trying to update the key value if and only if the current value is equal to the provided previous value parameter.

Response will look like:

```
{"node":
      {"children": {"empty" : true },
       "value"   : "Hi everyone",
       "key"     : "/greeting"
      },
```

```
 "action": "setNode"
}
```

HTTP response codes:

Code	Description
200	Node updated
201	New node created
409	CompareAndSet failed

5.1.3 DELETE Method

The delete method is used to delete the underlying configuration key, if the "**?recursive**" parameter is provided it also removes all keys which belong to the sub-tree of the given key.

Response will look like:

```
{"node":
      {"children": {"empty" : true },
       "value"   : "Hi everyone",
       "key"     : "/greeting"
      },
 "action": "deleteNode"
}
```

HTTP response codes:

Code	Description
200	Node deleted
404	Nodes does not exist
409	Node has children nodes, have to use "recursive" to delete.

5.2 Sessions

Service API that provides functionality to create and manage distributed sessions within cluster of distributed computational nodes. A session is a semi-permanent interactive information interchange, also known as a dialogue, a conversation or a meeting, between two or more communicating devices. A session is set up or established at a certain point in time, and then torn down at some later point. An established communication session required to define a context for membership and group API.

There are two endpoints: "**/{tenantId}/v1/sessions**" and "**/{tenantId}/v1/sessions/{clientSessionId}**", where:

tenantId – is the tenant identifier
clientSessionId– the identification of the client session

5.2.1 Endpoint "/{tenantId}/v1/sessions"

Supports HTTP GET and POST methods

5.2.1.1 GET Method

List all currently active clients session for a given tenant id.

Response will look like:

```
{"sessions":
      {"clientName": "someApplication",
       "sessionId"   : "1234567890",
       "key"     : "/greeting"
      },
 "action": "getSessions"
}
```

HTTP response codes:

Code	Description
200	OK
400	Bad request
404	Not Found
503	Service is not available

5.2.1.2 POST Method

Creates a session. The client is required to provide a unique name (clientName) and optionally a json object (as the body), which is any key/value data associated with the client (the metadata). The client must renew the session periodically to keep it active.

Expected HTTP body format:

```
{ "clientData":
            {"key1": "value1",
             "key2": "value2",
            },
  "clientName": "someApplication",
  "leaseSec": 600
}
```

Response will look like:

```
{"sessions":
      {"clientName": "someApplication",
```


```
        "sessionId"   : "1234567890",
        "key"     : "/greeting"
       },
  “action”: “createSession”
}
```


HTTP response codes:

Code	Description
201	Session created
400	Bad request
404	Not Found
409	Session already created
504	Service is not available

5.2.2 Endpoint “/{tenantId}/v1/sessions/{clientSessionId}”

Supports HTTP GET, DELETE and PUT methods

5.2.2.1 GET Method

Retrieves client metadata associated with clientSessionId.

Response will look like:


```
{"sessions":
       {"clientName": “someApplication”,
        "sessionId"   : "1234567890",
        "key"     : "/greeting"
       },
  “clientData”: {},
  “action”: “getData”
}
```


HTTP response codes:

Code	Description
200	OK
400	Bad request
404	Not Found
504	Service is not available

5.2.2.2 DELETE Method

Destroys client session.

HTTP response codes:

Code	Description
200	Session destroyed
400	Bad request
404	Not Found
504	Service is not available

5.2.2.3 PUT Method

Extends client session lease (renew session).

HTTP response codes:

Code	Description
200	OK
400	Bad request
404	Not Found
504	Service is not available

5.2.3 Groups

Service API that enables to manage groups of distributed nodes and provides the ability to create virtual clusters out of nodes, which form a group.

Exposes the following endpoints:

5.2.3.1 "/{tenantId}/v1/groups"

Supports **GET** HTTP method, which retrieves a list of existing group names for a given tenant id.

Response will look like:

```
{
 "groupView" :
          { "groupName" : "group1",
                          "clients" : [ {"clientName" : "client1",
            "sessionId" : "1234567890" } ],
            "size" : 123,
            "id" : 123456789   },
 "action" : "getGroups"
}
```

HTTP response codes:

Code	Description
200	OK
400	Bad request
404	Not Found
504	Service is not available

5.2.3.2 "/{tenantId}/v1/groups/{groupName}"

Supports the HTTP **GET** method, return group view: group id, group size and list of all members.

Response will look like:

```
{
 "groupView" :
          { "groupName" : "group1",
                                "clients" : [ {"clientName" : "client1",
            "sessionId" : "1234567890" } ],
            "size" : 123,
            "id" : 123456789   },
 "action" : "getGroupView"
}
```

HTTP response codes:

Code	Description
200	OK
400	Bad request
404	Not Found
504	Service is not available

5.2.3.3 "/{tenantId}/v1/groups/{groupName}/events"

Allows registering to any of a group's related events, such as: new member joined, member left and etc. Supports HTTP **GET** method.

You can specify the event of interest using a regular expression event.type.regexp. In this version, only GE_LEADER_ELECTED is supported for event.type.regexp. Every time the leader of the group changes GE_LEADER_ELECTED is fired. The registration will expire after a watch.timeout.sec (0 - means return immediately with a matching event). If an after.event.id is passed, the returned event is the first successive event that fits the reg-exp, otherwise the latest matching event is returned.

Response will look like:


```
{ "groupEvent" :
                    {"view" :
              { "groupName" : "group1",
                "clients" : [ { "clientName" : "client1",
                "sessionId" : "1234567890" } ],
                "size" : 123,
                "id" : 123456789
              },
        "id" : "1234567890",
        "type" : "GE_LEADER_ELECTED"
       },
 "action" : "getEvents"
}
```


HTTP response codes:

Code	Description
200	OK
400	Bad request
304	Not Modified
504	Service is not available

5.2.3.4 "/{tenantId}/v1/groups/{groupName}/leader"

Supports HTTP **GET** method, to retrieve the group leader. A leader is automatically elected by the system per group. You can register to an event to get notification on leader election or use this method to get the current elected leader.

Response will look like:


```
{
 "action" : "getLeader",
 "groupLeader" : {
              "groupName" : "group1",
              "client" : { "clientName" : "client1",
              "sessionId" : "aeiou"
              },
 "epoch" : 123456789
}
```


HTTP response codes:

Code	Description
200	OK
400	Bad request
404	Not Found
504	Service is not available

5.2.3.5 "/{tenantId}/v1/groups/{groupName}/sessions/{clientSessionsId}"

Supports HTTP **PUT** and **DELETE** methods. Provides the ability to join/create new group of distributed nodes and leave already existing group of nodes.

5.2.3.5.1 PUT Method

A group will be created if it doesn't exist already. You must have an active session to join a group and maintain membership.

HTTP response codes:

Code	Description
200	OK
400	Bad request
404	Not Found
504	Service is not available

5.2.3.5.2 DELETE Method

A group will be deleted if no members are left in the group.

HTTP response codes:

Code	Description
200	OK
400	Bad request
404	Not Found
504	Service is not available

6 DEPLOYMENT

In order to provide a highly available service we suggest the following deployment model. We suggest deploying 5 Frappe nodes across two DC, to be able to resist the failure of an entire DC and being able to provide a service. In front of the Frappe replica set nodes we use HAProxy to load balance traffic between Frappe instances and in order to support HAProxy high availability we would like to suggest using keepalived service to implement active IP takeover.

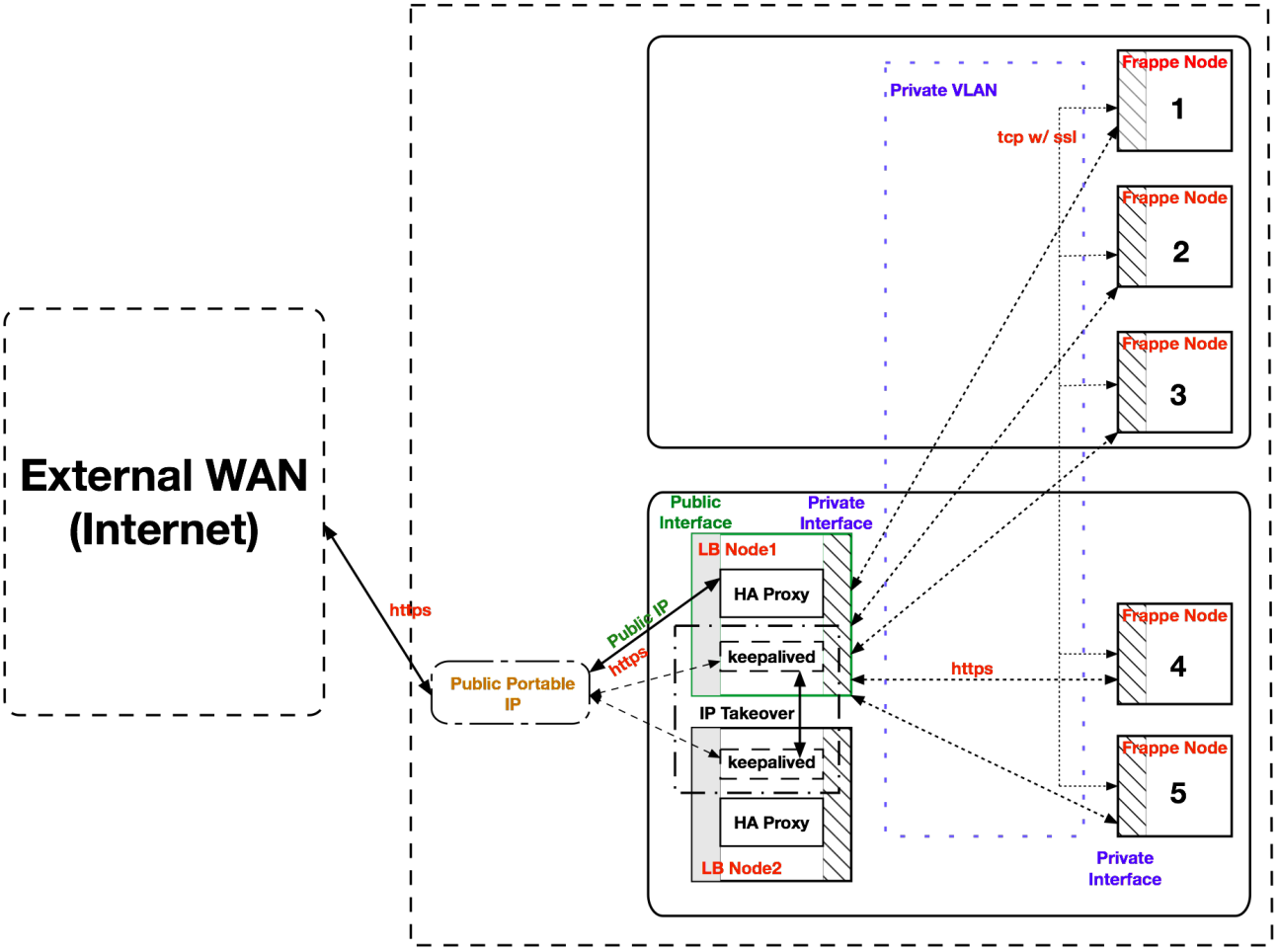

Figure 9: Deployment

7 CONCLUSION.

The key to the Frappe's efficiency is the new replicated state machine protocol, which is capable of avoiding the delays associated with the replication group reconfiguration by ordering commands speculatively in parallel to reaching agreement on the next stable configuration. We have discussed the Frappe's architecture, API, and core ideas underlying our speculative state machine implementation.

Frappe is being developed at IBM Research as core component of liberty profile application server to provide HA and cluster configuration management. During Panacea project it was extracted into the platform-as-a-service as a foundational tool, enabling to use it as a building block while providing HA and resilient cloud infrastructure. Integration of Frappe with Overlay Network coordination and management infrastructure allowed increasing stability and providing robustness and resilience of networking infrastructure.

8 REFERENCES.

[1] M. Burrows, "The Chubby lock service for loosely-coupled distributed systems," pp. 335–350, Nov. 2006.

[2] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "ZooKeeper: wait-free coordination for internet-scale systems," p. 11, Jun. 2010.

[3] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978.

[4] Y. Amir and C. Tutu, "From total order to database replication," in *Proceedings 22nd International Conference on Distributed Computing Systems*, 2002, pp. 494–503.

[5] F. B. Schneider, "Replication management using the state-machine approach," pp. 169–197, May 1993.

[6] N. A. Lynch and A. A. Shvartsman, "RAMBO: A Reconfigurable Atomic Memory Service for Dynamic Networks," pp. 173–190, Oct. 2002.

[7] M. K. Aguilera, I. Keidar, D. Malkhi, and A. Shraer, "Dynamic atomic storage without consensus," *J. ACM*, vol. 58, no. 2, pp. 1–32, Apr. 2011.

[8] L. Lamport, D. Malkhi, and L. Zhou, "Vertical paxos and primary-backup replication," in *Proceedings of the 28th ACM symposium on Principles of distributed computing - PODC ' 09*, 2009, p. 312.

[9] L. Lamport, "The part-time parliament," *ACM Trans. Comput. Syst.*, vol. 16, no. 2, pp. 133–169, May 1998.

[10] L. Lamport, D. Malkhi, and L. Zhou, "Reconfiguring a state machine," *ACM SIGACT News*, vol. 41, no. 1, p. 63, Mar. 2010.

[11] D. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kiciman, M. Merzbacher, D. Oppenheimer, N. Sastry, W. Tetzlaff, J. Traupman, and N. Treuhaft, "Recovery Oriented Computing (ROC): Motivation, Definition, Techniques," Mar. 2002.