Building the PaaS Cloud of the Future

Immigrant PaaS Technologies: Experimental Prototype of Software Components and Documentation

D7.3.3

Version 1.0

WP7 – Immigrant PaaS Technologies

Dissemination Level: Public

Lead Editor: Steve Strauch, University of Stuttgart

09/09/2013

Status: Final

The research leading to these results has received funding from the European Union's Seventh Framework Programme (FP7/2007-2013) under grant agreement n° 258862

Seventh Framework Programme

FP7-ICT-2009-5

Service and Software Architectures, Infrastructures and Engineering

Contributors:

Vasilios Andrikopoulos, University of Stuttgart
Nicolas Chabanoles, BONITASOFT
François Exertier, Bull
Rouven Krebs, SAP
Benoit Pelletier, Bull
Guillaume Porcher, Bull
Ricardo Jimenez-Peris, UPM
Simon Riggs, 2ndQ
Steve Strauch, University of Stuttgart
Johannes Wettinger, University of Stuttgart
Kathryn Woodcock, 2ndQ

Internal Reviewer(s):

Craig Sheridan, FLEXIANT
Michel Dao, FT

Version History

Version	Date	Authors	Sections Affected
0.1	07/08/2013	Johannes Wettinger (USTUTT), Steve Strauch (USTUTT)	Initial version containing structure and planned content
0.2	09/08/2012	Johannes Wettinger (USTUTT), Steve Strauch (USTUTT)	Content regarding Extension of Apache ServiceMix and partner responsibilities have been added.
0.3	04/09/2013	Steve Strauch (USTUTT)	Integration of contributions from involved WP7 partners
0.4	05/09/2013	Johannes Wettinger (USTUTT), Steve Strauch (USTUTT)	Final fixes before 4CaaSt internal review
0.5	08/09/2013	Craig Sheridan (FLEXIANT), Michel Dao (FT), Johannes Wettinger (USTUTT)	Integration of internal review feedback. Correction of grammar, spelling, and wording.
0.6	09/09/2013	Steve Strauch (USTUTT)	Preparation of final version and final fixes and corrections.

1.0	09/09/2013	Steve Strauch (USTUTT)	Final version

Table of Contents

List of Figures

List of Tables

Abbreviations

4CaaSt	Building the PaaS Cloud of the Future
ACID	Atomicity, Consistency, Isolation, Durability
API	Application Programming Interface
BC	Binding Component
BOS	Bonita Open Solution
EAR	Enterprise Archive
HDFS	Hadoop Distributed File System
ID	Identifier
JAR	Java Archive
JDBC	Java Database Connectivity
JDK	Java Development Kit
JOnAS	Java Open Application Server
JPA	Java Persistence API
OS	Operation System
OSGi	Open Services Gateway initiative framework
PGXS	PostgreSQL Extension System
RDBMS	Relational Database Management System
Repmgr	Replication Manager for PostgreSQL clusters
REST	Representational State Transfer
RP	Reporting Period
SA	Service Assembly
SoPeCo	Software Performance Cockpit
SUT	System Under Test
WAR	Web application ARchive
URL	Uniform Resource Locator
UUID	Universally Unique Identifiers
WP	Work Package

Executive Summary

The goal of work package 7 within 4CaaSt is to make proven immigrant platform technologies cloud-aware. Therefore, we are focusing on technologies stemming from the following four domains and which are reflected in the structure of the tasks accordingly: relational databases (Task 7.1), application servers (Task 7.2), composition frameworks and engines (Task 7.3), and integration technologies such as enterprise service bus (Task 7.4).

This deliverable and the corresponding prototypes provide the final versions of the prototypical implementation of WP7 components and their documentation. We concentrate in this document on the delta compared to D7.3.1 [1] and D7.3.2 [2] only and provide references to other deliverables and documents wherever possible.

In this document we focus on the information required for integrating and using the cloud-aware building blocks from WP7 within the 4CaaSt platform. In addition, we provide information on management and usage of the functionality extensions realized for each building block in RP3.

In particular, in this document we provide the prototype descriptions (Section 2), information on components management within the 4CaaSt platform such as deployment and monitoring (Section 3), and user guides (Section 4) for the following immigrant PaaS technologies extended for Cloud-awareness in RP2:

- *PostgreSQL* – a BiDirectional Replication for PostgreSQL.
- *CumuloNimbo* – an ultra-scalable database consisting of four main subsystems: transactional management, key-value data store, query engine, and configuration registry.
- *Java Open Application Server (JOnAS)* – an open source Java EE and OSGi application server extended for enabling interaction via REST interface and multi-tenant service handling.
- *Performance Isolation Benchmarking* – tooling to measure performance isolation
- *Bonita Open Solution* – an open source BPM solution extended for multi-tenancy regarding data and configuration isolation.
- *Orchestra* – an open source WS-BPEL engine.
- *Apache ServiceMix* – an open source Enterprise Service Bus extended for multi-tenant aware communication and multi-tenant aware administration and management.

1. Introduction

This document is based on the former WP7 deliverables D7.3.2 [2] and D7.2.3 [3]. Deliverable D7.3.2 reported and provided the second version of the WP7 prototypes and documentation focusing on functionality extension of WP7 building blocks. Based on this we provided new requirements focusing on the integration of WP7 components into the 4CaaSt platform in D7.2.3.

Thus, this document represents the documentation of the realization of the concrete requirements, specification, and design to achieve open functionality extensions required in order to achieve cloud-awareness of immigrant PaaS technologies, but mainly focusing on the integration of the cloud-aware immigrant technologies into the 4CaaSt platform compared to the prototypes delivered with D7.3.2. The corresponding runtime artefacts and components of the prototypes are referenced within this document and delivered together with this document.

1.1. Purpose and Scope

This is the last and final version of the prototypical implementations and documentations of the immigrant PaaS technologies to be made cloud-aware in WP7 and to be integrated into the 4CaaSt platform. This document provides the descriptions of the prototypes, information on how to integrate each of the prototypes into the 4CaaSt platform, and guidelines how to use each prototypes. Therefore, this document provides essential information enabling project internal usage, e.g., in WP8 concerning realization of 4CaaSt use cases, and project external usage of WP7 building blocks.

1.2. Document Overview

The remainder of this document is structured as follows: the prototype descriptions of the final version of WP7 components extended for Cloud-awareness and integrated into the 4CaaSt platform is presented in Section 2. The sequence of the components presentation in this document follows the order of the WP7 tasks the corresponding components belong to. Information on how to build, install, and set up the prototypical implementations focusing on the integration into the 4CaaSt platform is provided in Section 3. A user guide for each prototype focusing on the description of the graphical user interface and the API depending whether the corresponding prototype provides both or only one of them, is contained in Section 4. Finally, Section 5 concludes this document.

2. Prototype Description

In this section we present an overview of the final versions of the different WP7 building blocks focusing on the integration into the 4CaaSt platform and on the improvements and extensions that have been done during RP3.

2.1. PostgreSQL

In RP3, 4CaaST project has partially sponsored the development of a prototype for BiDirectional Replication (BDR) for PostgreSQL which has been started in RP2, see D7.2.3 [3] and D7.3.2. [2]. This is a more flexible implementation of the basic “multi-master database” concept. BDR is a feature being developed for inclusion in PostgreSQL core that provides greatly enhanced replication capabilities. BDR allows users to create a geographically distributed multi-master database using Logical Log Streaming Replication (LLSR) transport.

BDR is designed to provide both high availability and geographically distributed disaster recovery capabilities.

BDR is not “clustering” as some vendors use the term, in that it doesn't have a distributed lock manager, global transaction co-ordinator, etc. Each member server is separate yet connected, with design choices that allow separation between nodes that would not be possible with global transaction coordination.

The prototype achieves good performance and promises a good final implementation which is currently on-going. Details are included in the User Guide section.

PostgreSQL has been validated in the 4CaaSt scenario 8.1, see D8.1.4 [6].

2.2. CumuloNimbo

The CumuloNimbo ultra-scalable database consists of four main subsystems:

- Transactional management:
    - It is the key innovation in CumuloNimbo. It decomposes the ACID properties and scales each of them with a distributed scalable component in a composable manner.
- Key-value data store:
    - It provides scalable and reliable storage. It is provided by Apache HBase that runs on top of Hadoop Distributed File System (HDFS). HDFS provides persistency and reliability by means of replication.
- Query engine:
    - It provides full SQL to access the database.
- Configuration registry:
    - It provides a reliable registry where all configuration information is stored and updated during reconfigurations.

The different tiers are shown in Figure 1.

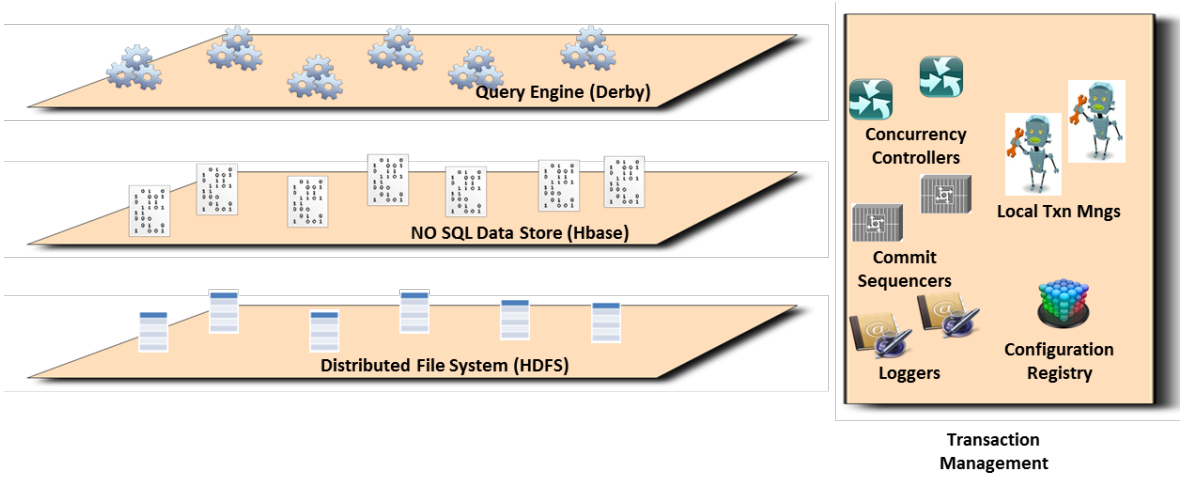

Figure 1. CumuloNimbo Ultra-Scalable Database Design

The way CumuloNimbo scales transactional processing is by decomposing the ACID properties and scaling each of them independently. Durability is provided by a set of loggers that enable to scale the bandwidth required for logging. Isolation is provided by concurrency controllers that detect conflicts among concurrent transactions and the snapshot server that provides a consistent snapshot to newly started transactions in the form of a start timestamp. Atomicity is guaranteed by local transaction managers and the commit sequencer. The commit sequencer determines the sequencing of transactions by assigning commit timestamps to committing update transactions.

CumuloNimbo can scale seamlessly thanks to the fact that transactions can commit in parallel without any synchronization, but full ACID consistency is guaranteed by only making visible consistent snapshots of the database. In this way, all the activities required to run and commit a transaction can be fully parallelized and the work distributed among several nodes. The commit sequencer and snapshot server are not parallelized since the amount of work they perform per transaction is so small that they are able to process around a million update transactions per second.

SQL processing is provided by SQL query engines. These query engines are stateless and only maintain client sessions and perform the query processing. The load of query processing can be distributed across as many query engines as needed. The query engines accept connections from the clients that connect through a JDBC driver. The JDBC driver discovers the available query engine instances via the registry where the current configuration is stored providing a list of all available components and their addresses.

Once discovered, the available query engine instantiates the JDBC Driver that connects to one of them randomly using an inverse probability to their relative load. The SQL requests are sent to the query engine. The query engine parses the SQL, generates the alternative query plans and the query optimizer chooses the one that looks more optimal. The query engine then executes this query plan. The leaves of the query plan are query operators that access the storage. They are custom storage operators that instead of accessing the local disk, they access the key-value data store to read, update, insert and delete tuples. In the current implementation the key-value data store used is HBase that runs on top of HDFS. HDFS provides persistency by storing the tables in persistent storage and high availability by replicating the files in which the tables are stored.

The relational schema is translated into a key-value data store schema. A table is created with the tuple values and associated to the primary key. Then, additional tables are created for secondary indexes to translate the secondary key into the primary key.

The client proxy of the key-value data store has extra functionality with respect to the original one from HBase. It has a wrapper that provides the transactional semantics. It also provides three extra methods to start, commit and abort transactions. The wrapper keeps private versions of updated data that are propagated to the key-value store after the transaction is made durable in the log. The read operations performed on behalf of a transaction on the key-value data store are adjusted with the private versions of the performed updates by the transaction.

2.3. Java Open Application Server – JOnAS

For this release we focused on the Java EE 6 Web profile certification and on the integration with the 4CaaSt platform's management facilities (monitoring, resource management, marketplace integration). The following sections outline the Java EE 6 Web profile implementation as well as the integration with the 4CaaSt platform.

2.3.1. Java EE 6 Web Profile Certification

JOnAS 5.3 has been enhanced to fulfil the Java EE 6 Web profile and get the certification from Oracle. See http://www.oracle.com/technetwork/java/javaee/overview/compatibility-jsp-136984.html.

JOnAS 5.3 now supports the following Java EE 6 Web profile specifications:

- Java Servlet 3.0
- JavaServer Faces (JSF) 2.0
- JavaServer Pages (JSP) 2.2
- Expression Language (EL) 2.2
- JavaServer Pages Standard Tag Library (JSTL) 1.2
- Debugging Support for Other Languages 1.0
- Enterprise JavaBeans (EJB) 3.1 Lite
- Java Persistence API (JPA) 2.0
- Contexts and Dependency Injection for Java 1.0
- Dependency Injection for Java 1.0
- Bean Validation 1.0
- Managed Beans 1.0
- Interceptors 1.1
- Common Annotations for the Java Platform 1.1

A part of the JOnAS internal components were upgraded and new ones were added (with the related glue) to implement the specifications. The major components used in the Java EE 6 Web profile are:

- Tomcat (Servlet, JSP)
- EasyBeans (EJB)
- Eclipselink (JPA)
- Jotm (JTA)
- Weld (CDI)

The certification process consisted in passing successfully 100% of the certification tests suite which represents about 23.000 tests.

2.3.2. JOnAS Integration With Resource Management

The resource management (WP4) is in charge of provisioning JOnAS instances within the 4CaaSt platform. The provisioning relies on Chef scripts (recipes) to automate the deployment and the configuration. A Chef cookbook "JOnAS_PIC" has been developed. Its purpose is both to deploy JOnAS as a platform instance component (PIC) as well as to deploy application components (ACs) on top of the AS. Different versions of this cookbook

were developed, allowing to deploy different versions of JOnAS. The latest version contains recipes allowing to deploy multi-tenant ACs and probes used to perform the accounting and charging.

2.3.3. JOnAS Integration With Marketplace

The JOnAS integration with marketplace consisted of:

- Providing a JOnAS blueprint
- Defining a pricing model

Both compliant to the approaches developed within 4CaaSt WPs 2 and 3.

2.3.4. JOnAS Integration With Accounting

The JOnAS product needs to be integrated with the accounting interface provided by WP5 to enable JOnAS to be offered in the 4CaaSt marketplace. This is used by the payment process for JOnAS instances.

The accounting for JOnAS is based on measuring the number of requests per tenant.

A JASMINe JMX probe is configured in the JOnAS Chef script to get the number of requests per tenant and a message is sent to the accounting API provided by WP5.

2.4. Performance Isolation Benchmarking

The Aspects concerning Performance Isolation benchmarks are manifold. In D7.3.2 [2] and D7.2.3 [3] we presented rather scientific contributions like the metrics to quantify performance isolation on the one hand and rather technical ones like the MT TPC-W Benchmark and a measurement environment on the other hand. In the following we recap the architectural aspects to provide a short description of the prototype. We also shortly recap the metrics.

2.4.1. Metrics

We presented two different approaches and three basic metrics, for quantifying performance isolation, decoupled from a concrete scenario. The first one is based on the impact of an increased workload, from one customer, on the QoS of other customers. This metric has strengths to express the impact of workload on the QoS which is relevant for capacity planning. The second group of metrics does reduce the workload of the customers working within their quota (W_a), if the workload of the disruptive customers increases. This maintains constant QoS for the residual workload of W_a. One subgroup of metrics relies on resulting significant points (e.g., when W_a becomes 0), another one on the area under the curve of W_a. The results show strengths of these metrics in ordering a system between the magnitudes of isolated and non-isolated which makes systems easily comparable.

2.4.2. Measurement Framework

Figure 2 describes the major components used for the measurements Framework. The SoPeCo controls the general process to estimate the performance isolation behaviour of the system under test. It selects possible workload configurations and quantifies the degree of isolation once all measurements are finished. The Experiment Definition defines meta information for the concrete measurement like the maximum duration or workload configurations used as reference for the measurements. The Multi-tenant measurements environment Controller is the plugged in functionality to connect the generic SoPeCo to the concrete environment. It maps the generic workload profiles from the SoPeCo to concrete settings used by a load driver which challenges the SUT. Furthermore, the Environment Controller collects the observed performance data and translates them into a format that could be understood by the SoPeCo specific plugins to measure performance isolation.

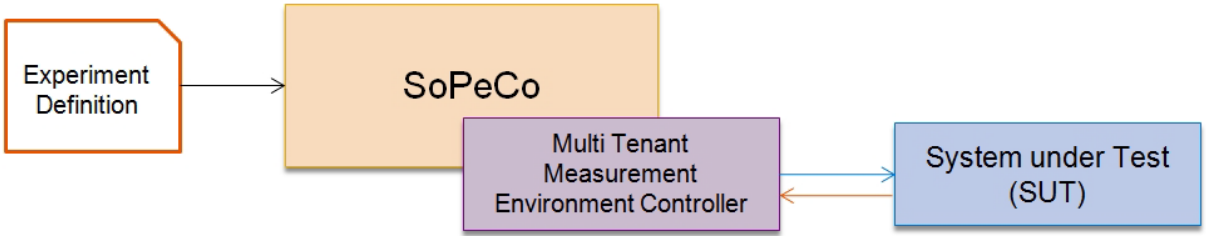

Figure 2. Performance Isolation Measurement Framework

2.4.3. MT TPC-W

The Multi-tenant version of the TPC-W consists of four major parts as shown in Figure 3. The Client Console presents the user interface to control the applications configuration and the load drivers load generation. The Load Driver consists of several component instances of concrete load drivers to generate load for the corresponding tenant. The DB Setup is a small tool which initializes the DB with data to ensure a consistent amount of data in the database for each run. The application itself (Book Store) is divided into two major parts. The Java servlets are rendering the Web page and containing application logic and the component managing the Persistency related activities. The Persistency encapsulates all DB interactions. The DB itself could be any SQL storage system with transaction capabilities.

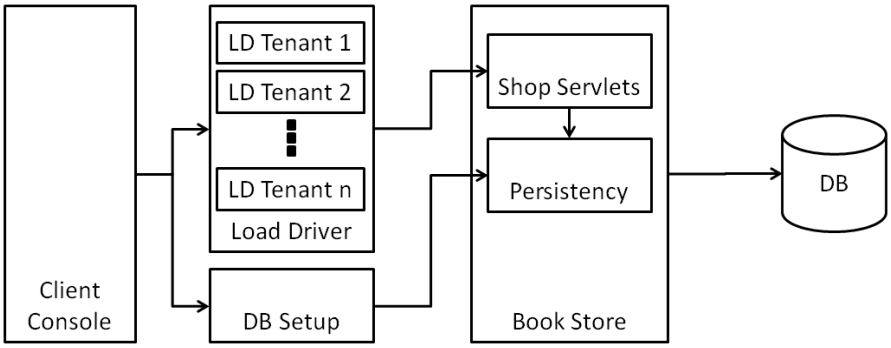

Figure 3. Multi-tenant Version of TPC-W

The Client Console, the DB Setup and the Load Driver is running on a J2SE environment. The Bookstore is also implemented in Java and relies on a servlet containing. The DB could be any SQL supporting environment.

2.5. Bonita Open Solution – BOS

In this section we introduce the architecture and realization of Bonita BPM 6 multi-tenant BPM solution. Moreover, we provide information on how it has been validated and evaluated in the scope of the 4CaaSt project.

The starting point of the realization was an early version of Bonita Open Solution v6 renamed Bonita BPM 6. The internal structure of Bonita is composed of several layers: API, BPM business logic, generic services.

The APIs are the entry points for an external customer layer to query (read / write) information from Bonita.

The BPM business logic is responsible for translating the customer query in BPM meaningful action and for delegating it to generic services.

An example of generic service is the persistence service that takes an object and stores it, another example is the cache service. For a given generic service multiple implementations

may exist. The implementation of a generic service is selected by configuration at deployment time.

For purposes of multi-tenancy, we introduced two different levels of users: system wide administrator and tenant-wide users. The system administrator is in charge of managing the deployment of Bonita, monitor the system resources consumption and is able to create / delete tenants. To ensure security this System administrator does not have access to internal tenant information. A tenant user is not aware of the existence of other tenants and cannot interfere with them. When creating a new tenant, the System admin provide a tenant user with the Tenant admin rights to allow him operate on the tenant: create / delete users, install / delete processes, etc.

Every single customer query may be attached to a tenant to ensure data isolation and accuracy. In order to make this possible we extended the APIs to let users identify themselves against a particular tenant user repository. By default this repository is the Bonita Database (e.g. PostgreSQL), but as any other service it may be possible to change the implementation of the service by the System admin at deployment time. After authentication each of the upcoming queries to API will be bound to the tenant thanks to the tenantId. This way the APIs will be able to gain access to the right instance of each service for the given tenant and ensure that data will be persisted at the right place, i.e. in the right tenant.

Adding new tenants has a cost in resource consumption (e.g. memory), that is why we had to allow cluster deployment combined with multi-tenancy. To keep deployment simple we've made the choice to have all nodes of the cluster idempotent; which means that all nodes have the same configuration and can do the same actions. Consequently each tenant exists on each node. A customer request may be redirected to any nodes of the cluster transparently and get the same answer. Of course all combinations of number of tenants and nodes are possible, e.g. 1/1, 1/n, n/1, etc.

The main challenges to support multiple active nodes in the cluster were the Cache Service and the Lock Service. The first one stores in memory temporary results that can be reused in order to improve response time of queries. The Lock service ensures the data consistency in case of concurrent access to shared resources.

On top of the APIs Bonita BPM comes with a Web UI, which allows the customization, administration, management, and interaction with the other layers. The Web UI offers a customizable interface for human and application interaction with the system, allowing for the administration and management of tenants and users. The API offers the same functionality as the Web UI, but also enables the integration and communication of external components and applications. A Java based client is provided to hide protocol complexity to call Bonita APIs.

Bonita BPM 6 has been validated by applying it to the 4CaaSt Tourism Application under the name Bonita Shop.

2.6. Orchestra

Although Orchestra was discontinued from the project in RP2 we had to integrate this component with the extension state shortly before the discontinuation in PR3, because Orchestra is used as cloud-aware building block in the 4CaaSt scenarios.

Thus, we addressed the following three domains with respect to integration into the 4CaaSt platform.

Orchestra was integrated with the Monitoring by defining KPIs designed to be collected by the JASMINe Monitoring Source as specified in WP5.

Orchestra was integrated with the Resource Management by providing a Chef cookbook for the Orchestra component (on top of JOnAS)

And finally, Orchestra was integrated with Marketplace by defining a Blueprint for this component.

2.7. Extension of Apache ServiceMix for Multi-Tenancy

In this section we introduce the architecture and realization of ESB^{MT} a multi-tenant open-source Enterprise Service Bus[1]. Moreover, we provide information on how this Cloud-aware building block has been validated and evaluated in the scope of the 4CaaSt project.

In the following we summarize the architecture and realization in details described in [1]. For purposes of implementing ESB^{MT} we extended the open source ESB Apache ServiceMix version 4.3.0 [5], hereafter referred to simply as ServiceMix. We present the realization of the components in the different layers of the framework in a bottom-up fashion, similarly to the presentation of the architecture in Figure 4.

The Resources layer consists of a *JBI Container Instance Cluster* and a set of registries. The JBI Container Instance Cluster bundles together multiple JBI. Each one of these instances (see Figure 5) performs the tasks usually associated with traditional ESB solutions, that is, message routing and transformation.

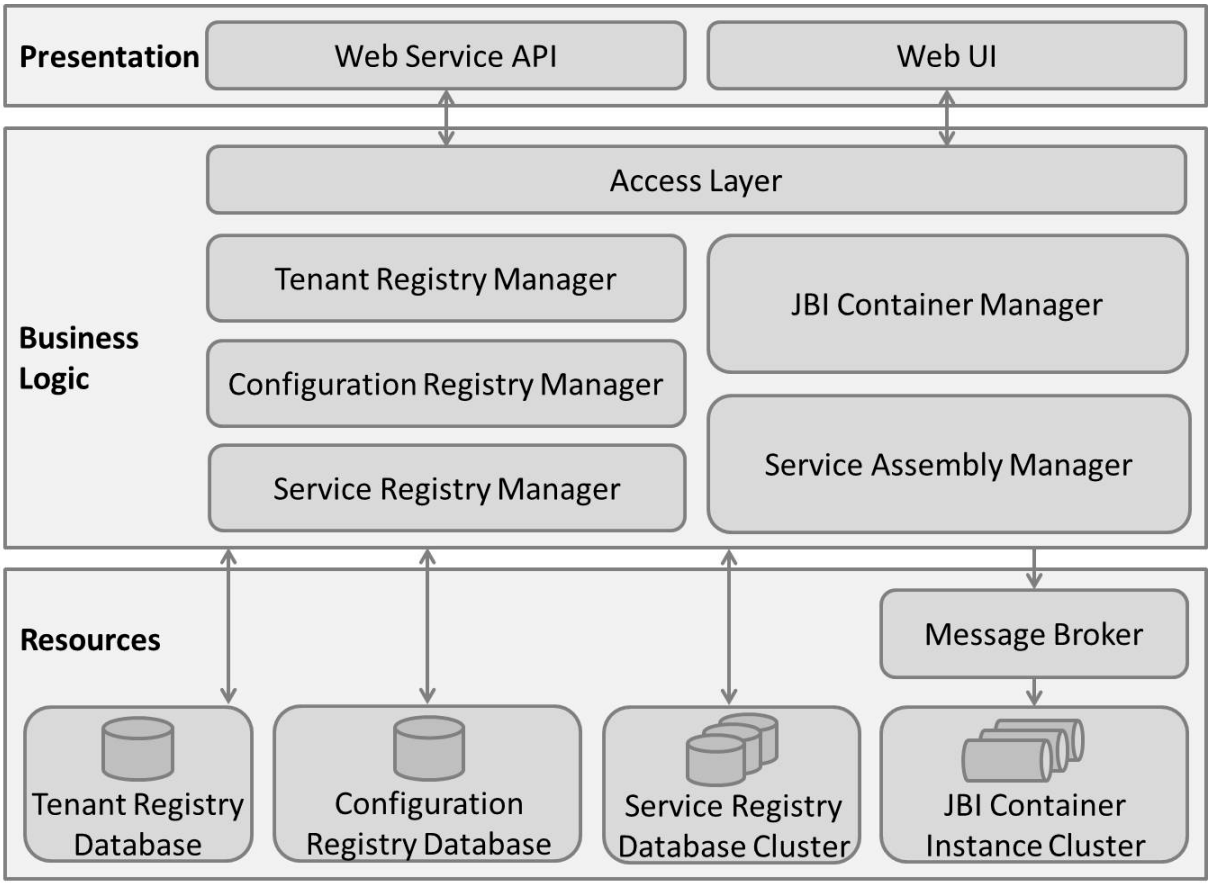

Figure 4. Overview of ESB^{MT}

[1] ESB^{MT}: http://www.iaas.uni-stuttgart.de/esbmt/

For purposes of performance, instances are organized in clusters, using an appropriate mechanism like the one offered by ServiceMix. Realizing multi-tenancy on this level means that both BCs and SEs are able to:

- handle service units and service assemblies containing tenant and user specific configuration information, and
- process such deployment artefacts accordingly in a multi-tenant manner. For example, a new tenant-specific endpoint has to be created whenever a service assembly is deployed to this JBI component in order to ensure data isolation between tenants.

The installation/uninstallation and configuration of BCs and SEs in a JBI Container Instance is performed through a set of standardized interfaces that also allow for backward compatibility with non multi-tenant aware components. In terms of implementation technologies, ServiceMix is based on the OSGi Framework [2]. OSGi bundles realize the ESB functionality complying to the JBI specification.

The original ServiceMix BC for HTTP version 2011.01 and the original Apache Camel SE version 2011.01 are extended in our prototype in order to support multi-tenant aware messaging. The Resources layer also contains three different types of registries (Figure 4): the *Service Registry* stores the services registered with the JBI environment, as well as the service assemblies required for the configuration of the BCs and SEs installed in each JBI Container Instance in the JBI Container Instance Cluster in a tenant-isolated manner; the *Tenant Registry* records the set of users for each tenant, the corresponding unique identifiers to identify them, as well all necessary information to authenticate them. Additionally, each tenant and user may have associated properties such as tenant or user name represented as key-value pairs. Moreover the password required for login before administration and configuration of the JBI environment is stored in the Tenant Registry represented as hash value generated with MD5. All these data are stored in a in a multi-tenant manner. The *ServiceRegistry*, *TenantRegistry*, and *ConfigurationRegistry* components are realized based on PostgreSQL version 9.1.1[3].

[2] OSGi Version 4.3: http://www.osgi.org/Download/Release4V43/

[3] PostgreSQL: http://www.postgresql.org/

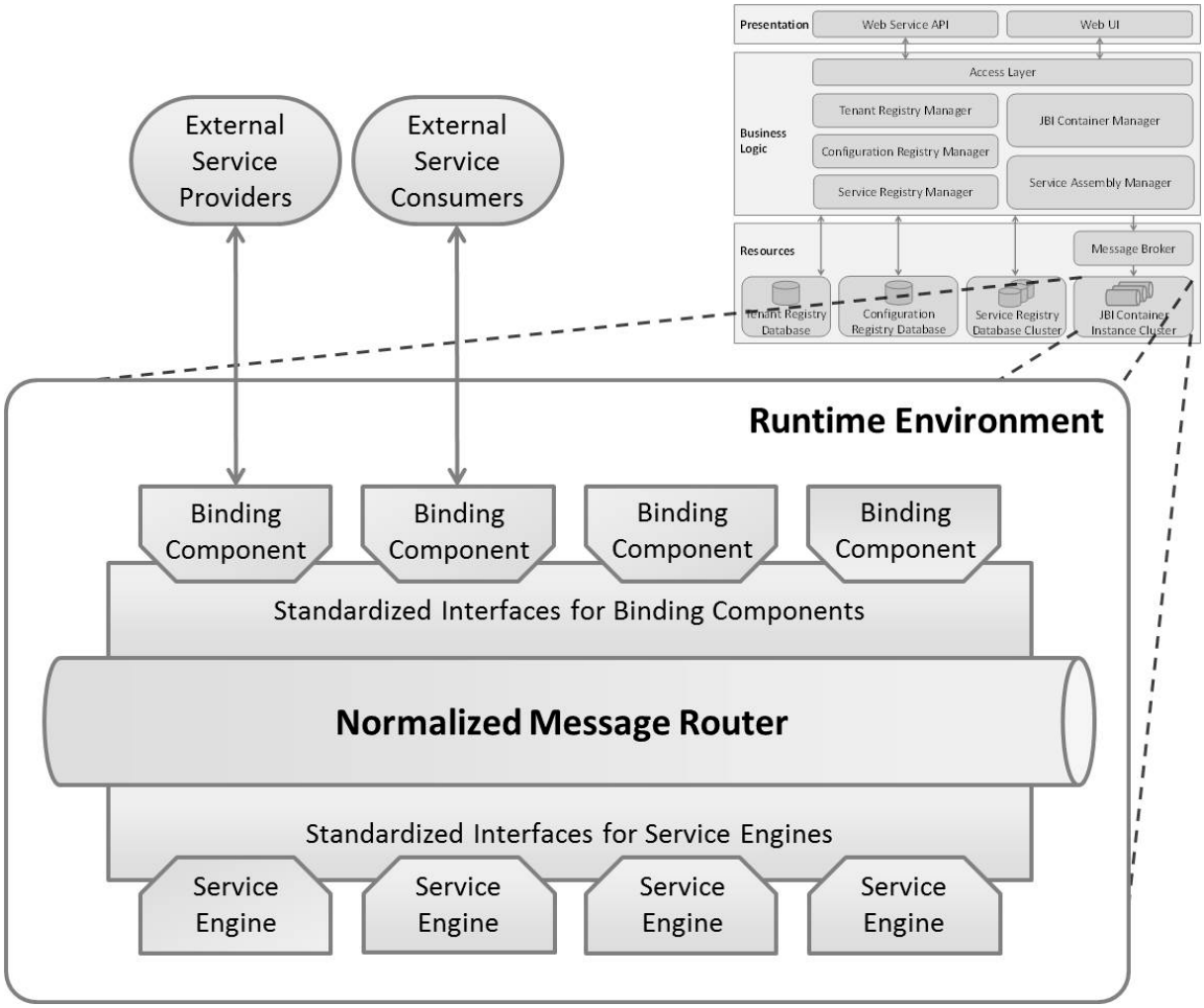

Figure 5. Architecture of an ESB Instance

The Business Logic layer contains an *Access Layer* component, which acts as a multi-tenancy enablement layer based on role-based access control. Different categories of roles can be defined based on their interaction with the system: system-level roles like administrators, and tenant-level roles like operators. The system administrator configures the whole system and assigns quotas of resource usage. Therefore, (s)he does not belong to any particular tenant, has unlimited permissions, and is allowed to interfere in the actions of all tenant users. The tenant users consume the quotas of resource usage to deploy service assemblies or to register services. This information is stored in the Configuration Registry. A tenant administrator can partition the quota of resource usage obtained from the system administrator. It is important that the system administrator assigns a default tenant administrator role to at least one tenant user to enable the corresponding tenant to perform actions. This default tenant administrator can then appoint other tenant administrators or assign tenant operator roles to tenant users.

The tenants and their corresponding users have to be identified and authenticated once when the interaction with the JBI environment is initiated. Afterwards, the authorized access is managed by the Access Layer transparently. The identification of tenants and users is performed based on unique *tenantID* and *userID* keys assigned to them by the Access Layer.

The various *Managers* in this layer (Figure 4) encapsulate the business logic required to manage and interact with the underlying components in the Resources layer: *Tenant Registry*, *Configuration Registry*, and *Service Registry Managers* for the corresponding registries, *JBI Container Manager* to install and uninstall BCs and SEs in JBI Containers in the cluster, and *Service Assembly Manager* for their configuration through deploying and undeploying appropriate service artifacts. The Business Logic layer of the proposed architecture is implemented as a Web application running in the Java EE 5 application server JOnAS version 5.2.2[4].

The Presentation layer contains the *Web UI* and the *Web service API* components which allow the customization, administration, management, and interaction with the other layers. The Web UI offers a customizable interface for human and application interaction with the system, allowing for the administration and management of tenants and users. The Web service API offers the same functionality as the Web UI, but also enables the integration and communication of external components and applications. It is realized based on the JAX-WS version 2.0.

ESB^{MT} has been validated by applying it to the 4CaaSt Taxi Application (scenario 8.1), which is described in detail in [6]. A video demonstrating the architecture of the Taxi Application as well as its functionality is available at http://www.iaas.uni-stuttgart.de/esbmt/#usecase.

The results of the performance evaluation of ESBMT are available in [1], [7].

[4] Java Open Application Server (JOnAS): http://jonas.ow2.org

3. Components Management

This chapter provides the information on component management within the 4CaaSt platform, e.g. regarding build, deployment, configuration, and start-up of each of the prototypical implementations of WP7 cloud-aware building blocks extended in RP3. Additionally, the links are provided, where the required software artefacts per component can be downloaded. The software artefacts are either publicly available, or they are available in the 4CaaSt project subversion repository.

3.1. PostgreSQL

As there have not been any changes of the Replication Manager (*repmgr*) with respect to the components management and integration into the 4CaaSt platform apart from some bug fixes, the interested reader is referred to D7.3.2 [2] for details.

3.2. CumuloNimbo

In this section we provide the information on how to integrate CumuloNimbo into the 4CaaSt platform.

3.2.1. Deployment

Three Chef cookbooks have been implemented following the 4CaaSt guidelines to deploy the CumuloNimbo ultra-scalable database. Their design and functionality has been described in D7.2.3 [3]. CumuloNimbo consists of several full subsystems including HDFS, HBase, Zookeeper, a modified Derby, and a set of transactional management components. Four conglomerates have been created to ease the deployment of CumuloNimbo and also to force the collocation of certain subsystems.

3.2.2. Monitoring and Accounting

Monitoring was not targeted as part of the integration exercise with 4CaaSt. The main reason is that it does not provide any added-value for 4CaaSt, and neither to CumuloNimbo that has its own monitoring subsystem.

Accounting was also not targeted since there is no special added value for 4CaaSt.

3.2.3. Integration with Marketplace

The integration with the marketplace is currently still on-going work.

3.3. Java Open Application Server – JOnAS

In this section we provide the information on how JOnAS is integrated into the 4CaaSt platform.

3.3.1. Deployment

Several cookbooks were developed to support the deployment of JOnAS on the 4CaaSt platform. Those cookbooks are available under http://109.231.68.170/4caast/trunk/WP4/Cookbooks/JOnAS_PIC-all/.

- Version 5.2, allows to deploy JOnAS 5.2.2 a fully certified Java EE 5 application server

- Version 5.3.0, allows to deploy JOnAS 5.3.0 a Java EE 6 Web Profile certified application server with Tomcat 6 as Web Container.
- Version 5.3.1, allows to deploy JOnAS 5.3.0 with Tomcat 7 as Web Container and the support for multi-tenancy

3.3.2. Monitoring and Accounting

A dedicated JASMINe Probe has been developed to publish accounting events to the accounting API provided by WP5. This probe is automatically deployed by Chef scripts when deploying a multi-tenant application on JOnAS.

The JASMINe Probe is available under http://109.231.68.170/4caast/trunk/WP5/jasmine-probe/jprobe-outer-4caast-accounting/trunk and Chef scripts under http://109.231.68.170/4caast/trunk/WP4/Cookbooks/JOnAS_PIC-all/5.3.1

3.3.3. Integration with Marketplace

The blueprint format for a Multi-tenant JOnAS 5.3 release is shown in the following listing.

```
<?xml version="1.0" encoding="UTF-8"?>
<bp:blueprint xmlns:bp="http://www.4caast.eu/blueprint" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
xsi:schemaLocation="http://www.4caast.eu/blueprint blueprint_template_v01.xsd
http://www.4caast.eu/MeteringSchema MeteringSchema_v03.xsd"
   xmlns:metering="http://www.4caast.eu/MeteringSchema">
   <bp:basic_properties_section>
      <bp:blueprint_id>JOnAS-Multi-Tenant-Servlet-Container</bp:blueprint_id>
      <bp:blueprint_name>JOnAS-Multi-Tenant-Servlet-Container</bp:blueprint_name>
      <bp:description>This is the Blueprint for Cloud-aware JOnAS Application server</bp:description>
      <bp:ownership>
         <bp:name>JOnAS Team</bp:name>
         <bp:uri>http://jonas.ow2.org</bp:uri>
      </bp:ownership>
      <bp:version>5.3.0-RC1</bp:version>
      <bp:release_date>2012-12-21</bp:release_date>

      <!-- multi tenant capable -->
      <bp:multi-tenant>true</bp:multi-tenant>

      <!--METERING INFO SECTION -->
      <bp:ext_property>
         <bp:p_name>metering info</bp:p_name>
         <metering:metering_section xmlns:metering="http://www.4caast.eu/MeteringSchema">
           <metering:metering_instructions_quantity>
             <metering:id>jonas-servlet-requests-by-webapp-by-tenant</metering:id>
             <metering:unit>CPUInstruction</metering:unit>
             <metering:unit_label>Number of requests by WebApp and by Tenant</metering:unit_label>
           </metering:metering_instructions_quantity>
         </metering:metering_section>
      </bp:ext_property>
      <!--END OF METERING SECTION -->
```

```
   </bp:basic_properties_section>

   <bp:offering_section>

      <bp:offering>

         <bp:offering_id>jonas-servlet-container-3.0</bp:offering_id>

         <bp:resource_name>servlet container v3.0</bp:resource_name>

         <bp:resource_type>Servlet Container v3.0</bp:resource_type>

         <bp:range_of_instance>

            <bp:minimum>1</bp:minimum>

            <bp:maximum>1</bp:maximum>

         </bp:range_of_instance>

      </bp:offering>

  </bp:offering_section>

</bp:blueprint>
```

3.4. Performance Isolation Benchmarking

3.4.1. Measurement Framework – Deployment and Installation

The implementation of the SoPeCo containing the Performance Isolation parts can be found at https://svn.forge.morfeo-project.org/4caast/trunk/WP7/release3.0/isolation . To deploy and install the generic Web based SoPeCo part one can follow the description at http://www.sopeco.org/tutorials/host-your-own-sopeco-instance. The described installation for R, a free software environment for statistical computing and graphics, and the description of the REST interfaces are not relevant for the 4CaaSt specific implementation and can be ignored. How to connect the SoPeCo to a concrete environment and how to develop a concrete environment connector is described in the User guide section as these steps have to be implemented by each user.

3.4.2. MT TPC-W

In the following we describe the installation of the MT TPC-W benchmark. To ensure a proper test scenario one needs 3 (DB and Application installed together) or 4 (DB and Application installed separately). To ensure a proper isolation of the application from the load driver and the client console, each element has to be installed on different machines. In the following we describe a concrete deployment with 4 hosts used in the context of 4CaaSt, see Figure 6.

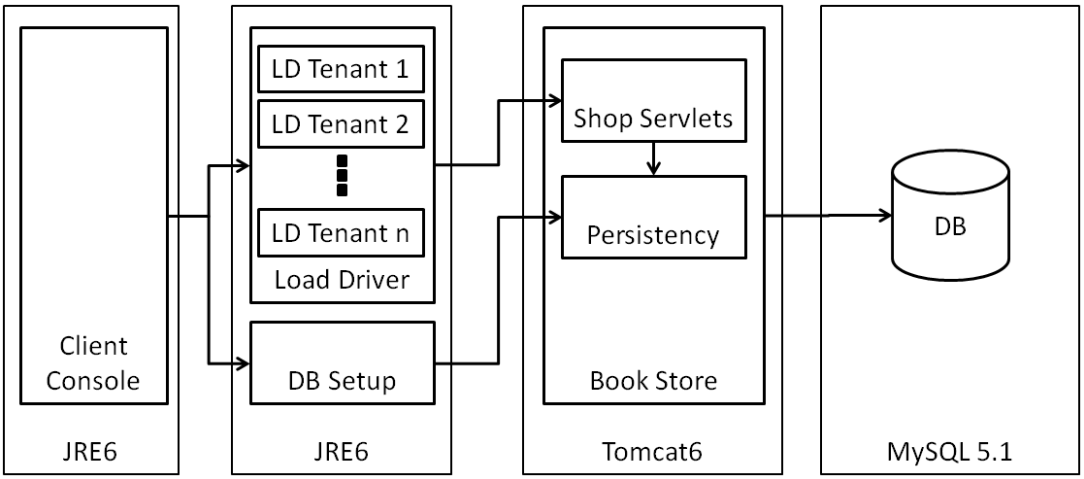

Figure 6. Deployment of MT TPC-W With Four Different Hosts

The client console, the Load Driver and DB Setup and the concrete Book Store can be downloaded from https://svn.forge.morfeo-project.org/4caast/trunk/WP7/release3.0/isolation. In the following we describe the concrete steps on how to deploy and start the components.

3.4.2.1. Client Console

The client console can be simply located at any arbitrary location on a host running JRE6. To start the Client Console one has to execute the ClientConsole.jar file with *java –jar ClientConsole.jar* . To configure the necessary connection information for the Load Driver and DB Setup (both listening at the same Port and IP behind a Facade). One has to enter the following information in the Client Console (> corresponds to output; < corresponds to input):

```
> Enter ip and port? Enter 'n' for default values. Press Enter to continue.

< Enter

> Enter ip of loaddriver:

< 10.55.12.152

> Enter port of loaddriver:

< 1099

> Connect to: 15.15.15.15:8585
```

3.4.2.2. Load Driver

The Load Driver can be located at any arbitrary location on a host running JRE6. Before starting ensure, that the java bin directory is added to the path environment variable. Simply start the Load Driver with `sh Start.sh` To stop the Load Driver execute `sh Stop.sh` . Both files are directly located by the jar file implementing the concrete functionality and located within the bundle to be downloaded at https://svn.forge.morfeo-project.org/4caast/trunk/WP7/release3.0/isolation

3.4.2.3. Book Store

The Book Store concrete war file can be downloaded at https://svn.forge.morfeo-project.org/4caast/trunk/WP7/release3.0/isolation. Simply drop the MTTPCWApplication.war into the Tomcats webapps folder to deploy the application. The DB has to be configured via the applications Web interface. Once the application runs call the URL http://<hostname>:<port>/MTTPCWApplication/CreateDatabase. At this page the concrete connection information can be entered. Once this is done click the link Delete Schema and Create Schema corresponding to the DB System used.

3.4.2.4. Database

For the installation of the Database simply follow the instructions of the corresponding Database Management System. Furthermore, after the installation create a database with the name *tpcw* and a user with read/write rights for this database with remote access.

3.5. Bonita Open Solution – BOS

In this section we focus on the management of Bonita BPM building block within the 4CaaSt platform including deployment, accounting, monitoring and integration with the 4CaaSt marketplace. All the artefacts required are available in the 4CaaSt svn at https://svn.forge.morfeo-project.org/4caast/trunk/WP7/release3.0/bonita.

The final version of the blueprint for Bonita is available at https://svn.forge.morfeo-project.org/4caast/trunk/WP7/release3.0/bonita/artefacts/blueprint/BonitaOpenSolution-v6.0-v04.xml. This blueprint builds the technical basis for offering the BPM Engine block within the 4CaaSt platform.

Chef Recipes are available on svn https://svn.forge.morfeo-project.org/4caast/trunk/WP7/release3.0/bonita/artefacts/deployment/.

Deployment with Chef has been tested on a Flexiscale Virtual machine with an Ubuntu OS.

In order to allow scripting over Bonita deployment to create tenants or manage applications, such as deploying new AC we had to develop a new Bonita Command Line Interface (CLI). This CLI can be called by DOS or Unix interpreters, e.g. batch, Bash. Consequently it is now possible to easily build Chef Recipes to create tenants on demand whenever the 4Caast Platform requires it.

For now this CLI is quite small but answers the need of 4Caast integration. It has been designed to be highly extensible. Consequently it is very easy to support any new use cases that may arise.

In the following listing the usage of the newly developed CLI is explained.

```
> ./bos-cli.sh help

Usage:

  -tenant

        -create  <tenantName>

        -enable  <tenantId>

        -disable <tenantId>

        -delete  <tenantId>


  -processInstance

        -start <processName:processVersion>


  -process

        -install <pathToBusinessArchive>

        -enable  <processId>

        -disable <processId>

        -delete  <processId>


  -user

        -create <user>

        -delete <userId>
```

For instance to deploy a new AC (e.g. BonitaShop 4Caast application in version 1.0) it is just a matter of executing the following command:

```
> ./bos-cli.sh -process -install /tmp/BonitaShop--1.0.bar
```

Find in the following listing the output produced when executing Chef scripts on a client to automatically deploy Bonita BPM using Chef Server and Chef Client.

```
> sudo chef-client
[Thu, 29 Aug 2013 09:40:53 +0000] INFO: Run List expands to [Bonita]
[Thu, 29 Aug 2013 09:40:53 +0000] INFO: Starting Chef Run for chef-client-bonita
[Thu, 29 Aug 2013 09:40:53 +0000] INFO: Loading cookbooks [Bonita]
[Thu, 29 Aug 2013 09:40:53 +0000] INFO: Storing updated cookbooks/Bonita/recipes/Undeploy_AC.rb in the
cache.
[Thu, 29 Aug 2013 09:40:53 +0000] INFO: Storing updated cookbooks/Bonita/recipes/Start_AC.rb in the cache.
[Thu, 29 Aug 2013 09:40:53 +0000] INFO: Storing updated cookbooks/Bonita/recipes/Stop_AC.rb in the cache.
[Thu, 29 Aug 2013 09:40:53 +0000] INFO: Storing updated cookbooks/Bonita/recipes/Start_PIC.rb in the cache.
[Thu, 29 Aug 2013 09:40:53 +0000] INFO: Storing updated cookbooks/Bonita/recipes/Deploy_PIC.rb in the
cache.
[Thu, 29 Aug 2013 09:40:53 +0000] INFO: Storing updated cookbooks/Bonita/recipes/Deploy_AC.rb in the cache.
[Thu, 29 Aug 2013 09:40:53 +0000] INFO: Storing updated cookbooks/Bonita/recipes/default.rb in the cache.
[Thu, 29 Aug 2013 09:40:53 +0000] INFO: Storing updated cookbooks/Bonita/recipes/Stop_PIC.rb in the cache.
[Thu, 29 Aug 2013 09:40:54 +0000] INFO: Storing updated cookbooks/Bonita/recipes/Undeploy_PIC.rb in the
cache.
[Thu, 29 Aug 2013 09:40:54 +0000] INFO: Storing updated cookbooks/Bonita/recipes/Add_Tenant.rb in the
cache.
[Thu, 29 Aug 2013 09:40:54 +0000] INFO: Processing directory[/opt/bonita] action create (Bonita::default
line 15)
[Thu, 29 Aug 2013 09:40:54 +0000] INFO: directory[/opt/bonita] created directory /opt/bonita
[Thu, 29 Aug 2013 09:40:54 +0000] INFO: Processing directory[/opt/bonita/CLI] action create
(Bonita::default line 20)
[Thu, 29 Aug 2013 09:40:54 +0000] INFO: directory[/opt/bonita/CLI] created directory /opt/bonita/CLI
[Thu, 29 Aug 2013 09:40:54 +0000] INFO: Processing remote_file[/tmp/BonitaBPMSubscription-6.0.0-Tomcat-
6.0.35.zip] action create_if_missing (Bonita::default line 26)
[Thu, 29 Aug 2013 09:41:07 +0000] INFO: remote_file[/tmp/BonitaBPMSubscription-6.0.0-Tomcat-6.0.35.zip]
updated
[Thu, 29 Aug 2013 09:41:07 +0000] INFO: Processing remote_file[/tmp/BOS-CLI-6.0.0-SP.zip] action
create_if_missing (Bonita::default line 32)
[Thu, 29 Aug 2013 09:41:07 +0000] INFO: remote_file[/tmp/BOS-CLI-6.0.0-SP.zip] updated
[Thu, 29 Aug 2013 09:41:07 +0000] INFO: Processing package[zip] action install (Bonita::default line 37)
[Thu, 29 Aug 2013 09:41:07 +0000] INFO: Processing execute[unzip] action run (Bonita::default line 40)
[Thu, 29 Aug 2013 09:41:07 +0000] INFO: execute[unzip] sh(unzip -u -o "/tmp/BonitaBPMSubscription-6.0.0-
Tomcat-6.0.35.zip"
)
Archive:  /tmp/BonitaBPMSubscription-6.0.0-Tomcat-6.0.35.zip
```



```
   creating: BonitaBPMSubscription-6.0.0-Tomcat-6.0.35/
   creating: BonitaBPMSubscription-6.0.0-Tomcat-6.0.35/webapps/
   creating: BonitaBPMSubscription-6.0.0-Tomcat-6.0.35/webapps/docs/
   creating: BonitaBPMSubscription-6.0.0-Tomcat-6.0.35/webapps/docs/architecture/
   creating: BonitaBPMSubscription-6.0.0-Tomcat-6.0.35/webapps/docs/architecture/requestProcess/
   inflating:
   [lines omitted for consistency]
[Thu, 29 Aug 2013 09:41:11 +0000] INFO: execute[unzip] ran successfully
[Thu, 29 Aug 2013 09:41:11 +0000] INFO: Processing execute[unzip] action run (Bonita::default line 49)
[Thu, 29 Aug 2013 09:41:11 +0000] INFO: execute[unzip] sh(unzip -u -o "/tmp/BOS-CLI-6.0.0-SP.zip"
)
Archive:  /tmp/BOS-CLI-6.0.0-SP.zip
   creating: bonita/
   creating: bonita/client/
   creating: bonita/client/conf/
  inflating: bos-cli.sh
  [lines ommitted for consistency]
[Thu, 29 Aug 2013 09:41:11 +0000] INFO: execute[unzip] ran successfully
[Thu, 29 Aug 2013 09:41:11 +0000] INFO: Processing execute[allow CLI execution] action run (Bonita::default
line 58)
[Thu, 29 Aug 2013 09:41:11 +0000] INFO: execute[allow CLI execution] sh(sudo chmod 777
/opt/bonita/CLI/*.sh)
[Thu, 29 Aug 2013 09:41:11 +0000] INFO: execute[allow CLI execution] ran successfully
[Thu, 29 Aug 2013 09:41:11 +0000] INFO: Processing remote_file[/opt/bonita/BonitaBPMSubscription-6.0.0-
Tomcat-6.0.35/bonita/server/licenses/BOSSP-6.0-Beta-NicolasChabanoles-chef-client-bonita-20130626-
20140626.lic] action create_if_missing (Bonita::default line 65)
[Thu, 29 Aug 2013 09:41:11 +0000] INFO: remote_file[/opt/bonita/BonitaBPMSubscription-6.0.0-Tomcat-
6.0.35/bonita/server/licenses/BOSSP-6.0-Beta-NicolasChabanoles-chef-client-bonita-20130626-20140626.lic]
updated
[Thu, 29 Aug 2013 09:41:11 +0000] INFO: Processing remote_file[/tmp/MuseumTicketBooking--1.0.bar] action
create_if_missing (Bonita::default line 72)
[Thu, 29 Aug 2013 09:41:11 +0000] INFO: remote_file[/tmp/MuseumTicketBooking--1.0.bar] updated
[Thu, 29 Aug 2013 09:41:11 +0000] INFO: Processing execute[start bundle] action run (Bonita::default line
78)
[Thu, 29 Aug 2013 09:41:11 +0000] INFO: execute[start bundle] sh(sudo sh /opt/bonita/BonitaBPMSubscription-
6.0.0-Tomcat-6.0.35/bin/catalina.sh start)
[Thu, 29 Aug 2013 09:41:11 +0000] INFO: execute[start bundle] ran successfully
[Thu, 29 Aug 2013 09:41:12 +0000] INFO: Chef Run complete in 18.491267 seconds
[Thu, 29 Aug 2013 09:41:12 +0000] INFO: Running report handlers
[Thu, 29 Aug 2013 09:41:12 +0000] INFO: Report handlers complete
```

3.6. Orchestra

In this section we provide the information on how Orchestra is integrated into the 4CaaSt platform.

3.6.1. Deployment

Several cookbooks were developed to support the deployment of Orchestra on the 4CaaSt platform. Those cookbooks are available under http://109.231.68.170/4caast/trunk/WP4/Cookbooks/JOnAS-Orchestra_PIC-all/.

3.6.2. Monitoring

Orchestra relies on JMX to provide monitoring indicators. The JASMINe Monitoring Source was used to collect the KPI as specified in WP5 [11], [12].

Four KPIs were defined to monitor Orchestra instances and to get information about the global status:

- NumberOfDeployedProcesses: number of processes deployed on the Orchestra instance
- AvgExecutionTime: average execution time of the processes deployed on the Orchestra instance
- NumberOfCompletedInstances: number of instances in completed state
- NumberOfRunningInstances : number of instances in running state

The Chef cookbook of Orchestra defines the KPIs and a JASMINe probe is configured at provisioning time to collect them as shown in the following listing.

```
normal["JOnAS_Orchestra_PIC"]["kpis"] = {
  "jmx_url" => "service:jmx:rmi://localhost/jndi/rmi://localhost:3098/jrmpconnector_jonas-orchestra",
  "kpis" => {
    "numberOfDeployedProcesses" => {"on" => "Orchestra:name=stats", "att" => "NumberOfDeployedProcesses"},
    "avgExecutionTime" => {"on" => "Orchestra:name=stats", "att" => "AvgExecutionTime"},
    "numberOfCompletedInstances"=>{"on" => "Orchestra:name=stats", "att" => "NumberOfCompletedInstances"},
    "numberOfRunningInstances" => {"on" => "Orchestra:name=stats", "att" => "NumberOfRunningInstances"}
  }
}
```

3.6.3. Accounting

This interface is not implemented as accounting is done at others levels in the 4CaaSt use cases.

3.6.4. Integration with Marketplace

An Orchestra blueprint was defined as shown in the following listing.

```
<?xml version="1.0" encoding="UTF-8"?>
<bp:blueprint xmlns:p="http://www.4caast.eu/blueprint"
              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
              xsi:schemaLocation="http://www.4caast.eu/blueprint blueprint_template_v01.xsd ">
   <bp:basic_properties_section>
      <bp:blueprint_id>jonas-orchestra-01-Blueprint</bp:blueprint_id>
      <bp:blueprint_name>jonas-orchestra-01-Blueprint</bp:blueprint_name>
```

```
      <bp:description>This is the Blueprint for Cloud-aware Orchestra BPEL engine embedded into JOnAS
         Application server </bp:description>
         <bp:ownership>
            <bp:name>JOnAS Team</bp:name>
            <bp:uri>Fehler! Hyperlink-Referenz ungültig.></bp:uri>
         </bp:ownership>
         <bp:version>4.9.0-M3</bp:version>
         <bp:release_date>2011-10-04</bp:release_date>
   </bp:basic_properties_section>
   <bp:offering_section>
      <bp:offering>
      <bp:offering_id>jonas-orchestra-01</bp:offering_id>
      <bp:resource_name>BPEL Compositon Engine</bp:resource_name>
      <bp:resource_type>BPEL Compositon Engine</bp:resource_type>
   </bp:offering>
   </bp:offering_section>
</bp:blueprint>
```

3.7. Extension of Apache ServiceMix for Multi-Tenancy

In this section we focus on the management of the EBM^{MT} building block within the 4CaaSt platform including deployment, accounting, monitoring and integration with the 4CaaSt marketplace. As the deployment, accounting, and monitoring is not 4CaaSt specific and thus does also work without the 4CaaSt platform we have created a manual providing a step-by-step tutorial for deployment and configuration, accounting, and monitoring of ESB^{MT} available at http://www.iaas.uni-stuttgart.de/esbmt/#installation. The tutorial also providing the links to all the publicly available artefacts required is in addition also available in the 4CaaSt svn at https://svn.forge.morfeo-project.org/4caast/trunk/WP7/release3.0/esb/ESB_MT_manual.pdf.

As this tutorial covers deployment, monitoring, and accounting in the following we describe the integration with the 4CaaSt marketplace, which requires the availability of other 4CaaSt components such as the Blueprint Editor and the Price Model Editor. The integration of ESB^{MT} with the 4CaaSt marketplace includes two aspects: the creation of a blueprint specifying the technical details and the definition of a price model and product.

The final version of the blueprint for ESB^{MT} is available at https://svn.forge.morfeo-project.org/4caast/trunk/WP7/release3.0/esb/artefacts/blueprint/blueprint_asm.xml and has been created using the latest version of the Blueprint Editor developed by WP2 [8]. This blueprint builds the technical basis for offering the multi-tenant ESB as building block within the 4CaaSt platform.

The final version of the price model for ESB^{MT} is available at https://svn.forge.morfeo-project.org/4caast/trunk/WP7/release3.0/esb/artefacts/price_model/pricemodel_asm.xml and has been created using the Price Model Editor developed by WP3 [9].

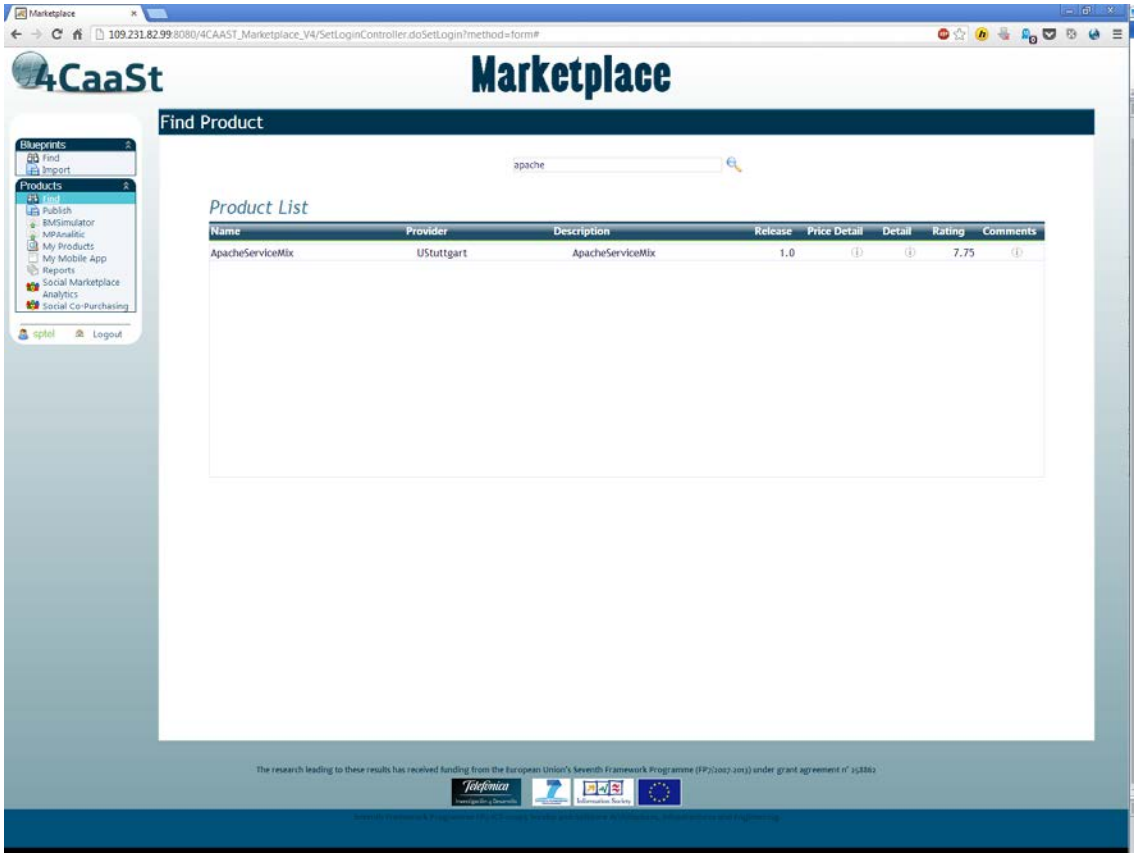

Figure 7. Screenshot of 4CaaSt Marketplace Showing ESB^{MT} Product

Figure 7 presents a screenshot of the Marketplace showing the product of ESB^{MT}.

4. User Guide

This chapter provides information and updates on the user guidelines provided in D7.3.2 [2].

4.1. PostgreSQL

In the following we provide guidelines on the usage of PostgreSQL repmgr.

Guidance on getting a testing setup established can be found in Section 4.1.1.5. Please read the full documentation if you intend to put Bi-Directional Replication into production.

4.1.1. Logical Log Streaming Replication

Logical log streaming replication (LLSR) allows one PostgreSQL master (the "upstream master") to stream a sequence of changes to another read/write PostgreSQL server (the "downstream master"). Data is sent in one direction only over a normal libpq connection.

Multiple LLSR connections can be used to set up bi-directional replication as discussed later in this guide.

4.1.1.1. Overview of Logical Replication

In some ways LLSR is similar to "streaming replication" i.e. physical log streaming replication (PLSR) from a user perspective; both replicate changes from one server to another. However, in LLSR the receiving server is also a full master database that can make changes, unlike the read-only replicas offered by PLSR hot standby. Additionally, LLSR is per-database, whereas PLSR is per-cluster and replicates all databases at once. There are many more differences discussed in the relevant sections of this document.

In LLSR the data that is replicated is change data in a special format that allows the changes to be logically reconstructed on the downstream master. The changes are generated by reading transaction log (WAL) data, making change capture on the upstream master much more efficient than trigger based replication, hence why we call this "logical log replication". Changes are passed from upstream to downstream using the libpq protocol, just as with physical log streaming replication.

One connection is required for each PostgreSQL database that is replicated. If two servers are connected, each of which has 50 databases then it would require 50 connections to send changes in one direction, from upstream to downstream. Each database connection must be specified, so it is possible to filter out unwanted databases simply by avoiding configuring replication for those databases.

Setting up replication for new databases is not (yet) automatic, so additional configuration steps are required after *CREATE DATABASE*. A restart of the downstream master is also required. The upstream master only needs restarting if the *max_logical_slots* parameter is too low to allow a new replica to be added. Adding replication for databases that do not exist yet will cause an ERROR, as will dropping a database that is being replicated. Setup is discussed in more detail below.

Changes are processed by the downstream master using BDR plug-ins. This allows flexible handing of replication input, including:

- BDR apply process - applies logical changes to the downstream master. The apply process makes changes directly rather than generating SQL text and then parse/plan/executing SQL.
- Textual output plugin - a demo plugin that generates SQL text (but doesn't apply changes)
- pg_xlogdump - examines physical WAL records and produces textual debugging output. This server program is included in PostgreSQL 9.3.

4.1.1.2. Replication of DML Changes

All changes are replicated: *INSERT*, *UPDATE*, *DELETE,* and *TRUNCATE*. (TRUNCATE is not yet implemented, but will be implemented before the feature goes to final release).

Actions that generate WAL data but don't represent logical changes do not result in data transfer, e.g. full page writes, VACUUMs, hint bit setting. LLSR avoids much of the overhead from physical WAL, though it has overheads that mean that it doesn't always use less bandwidth than PLSR.

Locks taken by *LOCK* and *SELECT ... FOR UPDATE/SHARE* on the upstream master are not replicated to downstream masters. Locks taken automatically by *INSERT*, *UPDATE*, *DELETE* or *TRUNCATE* *are* taken on the downstream master and may delay replication apply or concurrent transactions – see Section 4.1.2.3.1.

TEMPORARY and *UNLOGGED* tables are not replicated. In contrast to physical standby servers, downstream masters can use temporary and unlogged tables. However, temporary tables remain specific to a particular session so creating a temporary table on the upstream master does not create a similar table on the downstream master.

DELETE and UPDATE statements that affect multiple rows on upstream master will cause a series of row changes on downstream master. These are likely to go at same speed as on origin, as long as an index is defined on the Primary Key of the table on the downstream master. *UPDATE*s and *DELETE*s require some form of unique constraint, either *PRIMARY KEY* or *UNIQUE NOT NULL*. A warning is issued in the downstream master's logs if the expected constraint is absent. *INSERT* on upstream master do not require a unique constraint in order to replicate correctly, though such usage would prevent conflict detection between multiple masters, if that was considered important.

*UPDATE*s that change the value of the Primary Key of a table will be replicated correctly.

The values applied are the final values from the *UPDATE* on the upstream master, including any modifications from before-row triggers, rules or functions. Any reflexive conditions, such as N = N+ 1 are resolved to their final value. Volatile or stable functions are evaluated on the master side and the resulting values are replicated. Consequently any function side-effects (writing files, network socket activity, updating internal PostgreSQL variables, etc.) will not occur on the replicas as the functions are not run again on the replica.

All columns are replicated on each table. Large column values that would be placed in TOAST tables are replicated without problem, avoiding de-compression and re-compression. If we update a row but do not change a TOASTed column value, then that data is not sent downstream.

All data types are handled, not just the built-in datatypes of PostgreSQL core. The only requirement is that user-defined types are installed identically in both upstream and downstream master (see "Limitations").

The current LLSR plugin implementation uses the binary libpq protocol, so it requires that the upstream and downstream master use same CPU architecture and word-length, i.e. "identical servers", as with physical replication. A textual output option will be added later for passing data between non-identical servers, e.g. laptops or mobile devices communicating with a central server.

Changes are accumulated in memory (spilling to disk where required) and then sent to the downstream server at commit time. Aborted transactions are never sent. Application of changes on downstream master is currently single-threaded, though this process is efficiently implemented. Parallel apply is a possible future feature, especially for changes made while holding *AccessExclusiveLock*.

Changes are applied to the downstream master in the sequence in which they were commited on the upstream master. This is a known-good serialized ordering of changes, so replication serialization failures are not theoretically possible. Such failures are common in systems that use statement based replication (e.g. MySQL) or trigger based replication (e.g. Slony version 2.0). Users should note that this means the original order of locking of tables is

not maintained. Although lock order is probably not an issue for the set of locks held on upstream master, additional locking on downstream side could cause lock waits or deadlocking in some cases, discussed in further detail later in this section.

Larger transactions spill to disk on the upstream master once they reach a certain size. Currently, large transactions can cause increased latency. Future enhancement will be to stream changes to downstream master once they fill the upstream memory buffer, though this is likely to be implemented in version 9.5.

SET statements and parameter settings are not replicated. This has no effect on replication since we only replicate actual changes, not anything at SQL statement level. We always update the correct tables, whatever the setting of *search_path*. Values are replicated correctly irrespective of the values of *bytea_output*, *TimeZone*, *DateStyle*, etc.

NOTIFY is not supported across log based replication, either physical or logical. *NOTIFY* and *LISTEN* will work fine on the upstream master but an upstream *NOTIFY* will not trigger a downstream *LISTEN*er.

In some cases, additional deadlocks can occur on apply. This causes an automatic retry of the apply of the replaying transaction and is only an issue if the deadlock recurs repeatedly, delaying replication.

From a performance and concurrency perspective the BDR apply process is similar to a normal backend. Frequent conflicts with locks from other transactions when replaying changes can slow things down and thus increase replication delay, so reducing the frequency of such conflicts can be a good way to speed things up. Any lock held by another transaction on the downstream master - *LOCK* statements, *SELECT ... FOR UPDATE/FOR SHARE*, or *INSERT/UPDATE/DELETE* row locks - can delay replication if the replication apply process needs to change the locked table/row.

4.1.1.3. Table Definitions and DDL Replication

DML changes are replicated between tables with matching *"Schemaname"*.*"Tablename"* on both upstream and downstream masters. E.g. changes from upstream's *public.mytable* will go to downstream's *public.mytable* while changes to the upstream *mychema.mytable* will go to the downstream *myschema.mytable*. This works even when no schema is specified on the original SQL since we identify the changed table from its internal OIDs in WAL records and then map that to whatever internal identifier is used on the downstream node.

This requires careful synchronization of table definitions on each node otherwise *ERROR*s will be generated by the replication apply process. In general, tables must be an exact match between upstream and downstream masters.

There are no plans to implement working replication between dissimilar table definitions. Tables must meet the following requirements to be compatible for purposes of LLSR:

- The downstream master must only have constraints (*CHECK*, *UNIQUE*, *EXCLUSION*, *FOREIGN KEY*, etc.) that are also present on the upstream master. Replication may initially work with mismatched constraints but is likely to fail as soon as the downstream master rejects a row the upstream master accepted.
- The table referenced by a FOREIGN KEY on a downstream master must have all the keys present in the upstream master version of the same table.
- Storage parameters must match except for as allowed below
- Inheritance must be the same
- Dropped columns on master must be present on replicas
- Custom types and enum definitions must match exactly

- Composite types and enums must have the same oids on master and replication target
- Extensions defining types used in replicated tables must be of the same version or fully SQL-level compatible and the oids of the types they define must match.

The following differences are permissible between tables on different nodes:

- The table's *pg_class* oid, the oid of its associated TOAST table, and the oid of the table's rowtype in *pg_type* may differ;
- Extra or missing non-*UNIQUE* indexes
- Extra keys in downstream lookup tables for *FOREIGN KEY* references that are not present on the upstream master
- The table-level storage parameters for fillfactor and autovacuum
- Triggers and rules may differ (they are not executed by replication apply)

Replication of DDL changes between nodes will be possible using event triggers, but is not yet integrated with LLSR (see Section 4.1.1.4).

Triggers and Rules are NOT executed by apply on downstream side, equivalent to an enforced setting of *session_replication_role* = *origin*.

In future it is expected that composite types and enums with non-identical oids will be converted using text output and input functions. This feature is not yet implemented.

4.1.1.4. LLSR Limitations

The current LLSR implementation is subject to some limitations, which are being progressively removed as work progresses.

4.1.1.4.1. Data Definition Compatibility

Table definitions, types, extensions, etc must be near identical between upstream and downstream masters. See Section 4.1.1.3.

4.1.1.4.2. DDL Replication

DDL replication is not yet supported. This means that any *ALTER TABLE* may cause the definitions of tables on either end of a link to go out of sync, causing replication to fail.

DROP TABLE of a table on a downstream master or BDR member may cause replication to halt as pending rows for that table cannot be applied.

CREATE TABLE will work without problems, but the table must be created on all nodes before rows are inserted on any node, otherwise replication issues will arise.

The *PRIMARY KEY* of a table may not be dropped. Other indexes may be added and removed freely as they do not affect replication.

4.1.1.4.3. Upstream Feedback

No feedback from downstream masters to the upstream master is implemented for asynchronous LLSR, so upstream masters must be configured to keep enough WAL. See Section 4.1.1.10.

4.1.1.4.4. TRUNCATE is not Replicated

TRUNCATE is not yet supported, however workarounds with user-level triggers are possible and a ProcessUtility hook is planned to implement a similar approach globally. The safest option is to define a user-level BEFORE trigger on each table that RAISEs an ERROR when TRUNCATE is attempted.

A simple truncate-blocking trigger is shown in the following listing:

```
CREATE OR REPLACE FUNCTION deny_truncate() RETURNS trigger AS $$

 BEGIN

   IF tg_op = 'TRUNCATE' THEN

     RAISE EXCEPTION 'TRUNCATE is not supported on this table. Please use DELETE FROM.';

   ELSE

     RAISE EXCEPTION 'This trigger only supports TRUNCATE';

   END IF;

 END;

 $$ LANGUAGE plpgsql;
```

It can be applied to a table with:

```
CREATE TRIGGER deny_truncate_on_<tablename> BEFORE TRUNCATE ON <tablename>

 FOR EACH STATEMENT EXECUTE PROCEDURE deny_truncate();
```

A PL/PgSQL DO block that queries *pg_class* and loops over it to *EXECUTE* a dynamic SQL *CREATE TRIGGER* command for each table that does not already have the trigger can be used to apply the trigger to all tables.

Alternately, there will be a *ProcessUtility_hook* available in the BDR extension to automatically prevent unsupported operations like *TRUNCATE*.

4.1.1.4.5. CLUSTER and VACUUM FULL limitations

CLUSTER and *VACUUM FULL* should now be supported without limitation.

4.1.1.5. Initial Setup

To set up LLSR or BDR you first need a patched PostgreSQL that can support LLSR/BDR, then you need to create one or more LLSR/BDR senders and one or more LLSR/BDR receivers.

4.1.1.5.1. Installing the Patched PostgreSQL Binaries

Currently BDR is only available in builds of the 'bdr' branch on Andres Freund's Git repo on git.postgresql.org. PostgreSQL 9.2 and below do not support BDR, and 9.3 requires patches, so this guide will not work for you if you are trying to use a normal install of PostgreSQL.

First you need to clone, configure, compile and install like normal. Clone the sources from *git://git.postgresql.org/git/users/andresfreund/postgres.git* and checkout the *bdr* branch.

If you have an existing local PostgreSQL git tree specify it as *--reference /path/to/existing/tree* to greatly speed your git clone.

The following listing shows an example.

```
mkdir -p $HOME/bdr

 cd bdr

 git clone -b bdr git://git.postgresql.org/git/users/andresfreund/postgres.git $HOME/bdr/postgres-bdr-src

 cd postgres-bdr-src

 ./configure --prefix=$HOME/bdr/postgres-bdr-bin

 make install

 (cd contrib/btree_gist && make install)

 (cd contrib/bdr && make install)
```


This will put everything in *$HOME/bdr*, with the source code and build tree in *$HOME/bdr/postgres-bdr-src* and the installed PostgreSQL in *$HOME/bdr/postgres-bdr-bin*. This is a convenient setup for testing and development because it doesn't require you to set up new users, wrangle permissions, run anything as root, etc. but it isn't recommended that you deploy this way in production.

To actually use these new binaries you will need to export *PATH=$HOME/bdr/postgres-bdr-bin/bin:$PATH* before running *initdb*, *postgres*, etc. You don't have to use the *psql* or *libpq* you compiled but you're likely to get version mismatch warnings if you don't.

4.1.1.6. Parameter Reference

The following parameters are new or have been changed in PostgreSQL's new logical streaming replication.

4.1.1.6.1. shared_preload_libraries = 'bdr'

To load support for receiving changes on a downstream master, the *bdr* library must be added to the existing 'shared_preload_libraries' parameter. This loads the bdr library during postmaster start-up and allows it to create the required background worker(s).

Upstream masters don't need to load the bdr library unless they're also operating as a downstream master as is the case in a BDR configuration.

4.1.1.6.2. bdr.connections

A comma-separated list of upstream master connection names is specified in *bdr.connections*. These names must be simple alphanumeric strings. They are used when naming the connection in error messages, configuration options and logs, but are otherwise of no special meaning. A typical two-upstream-master setting might be: bdr.connections = 'upstream1, upstream2'

4.1.1.6.3. bdr.<connection_name>.dsn

Each connection name must have at least a data source name specified using the *bdr.<connection_name>.dsn* parameter. The DSN syntax is the same as that used by libpq so it is not discussed in further detail here. A *dbname* for the database to connect to must be specified; all other parts of the DSN are optional.

The local (downstream) database name is assumed to be the same as the name of the upstream database being connected to, though future versions will make this configurable.

For the above two-master setting for *bdr.connections* the DSNs might look like:

bdr.upstream1.dsn = 'host=10.1.1.2 user=postgres dbname=replicated_db'

bdr.upstream2.dsn = 'host=10.1.1.3 user=postgres dbname=replicated_db'

4.1.1.6.4. bdr.synchronous_commit

This boolean option controls the *synchronous_commit* setting for BDR apply workers. It defaults to *on*.

If set to *off*, BDR apply workers will perform async commits, allowing PostgreSQL to considerably improve throughput. It is safe unless you intend to run BDR with synchronous replication, in which case *bdr.synchronous_commit* must be left *on*.

4.1.1.6.5. max_logical_slots

The new parameter *max_logical_slots* has been added for use on both upstream and downstream masters. This parameter controls the maximum number of logical replication slots - upstream or downstream - that this cluster may have at a time. It must be set at postmaster start time.

As logical replication slots are persistent, slots are consumed even by replicas that are not currently connected. Slot management is discussed in Starting, Stopping and Managing Replication.

max_logical_slots should be set to the sum of the number of logical replication upstream masters this server will have plus the number of logical replication downstream masters will connect to it it.

4.1.1.6.6. wal_level = 'logical'

A new setting, *'logical'*, has been added for the existing *wal_level* parameter. *'logical'* includes everything that the existing *hot_standby* setting does and adds additional details required for logical changeset decoding to the write-ahead logs.

This additional information is consumed by the upstream-master-side xlog decoding worker. Downstream masters that do not also act as upstream masters do not require *wal_level* to be increased above the default *'minimal'*.

wal_level, except for the new *'logical'* setting, is http://www.postgresql.org/docs/current/static/runtime-config-wal.html documented in the main PostgreSQL manual.

4.1.1.6.7. max_wal_senders

Logical replication hasn't altered the *max_wal_senders* parameter, but it is important in upstream masters for logical replication and BDR because every logical sender consumes a *max_wal_senders* entry.

You should configure *max_wal_senders* to the sum of the number of physical and logical replicas you want to allow an upstream master to serve. If you intend to use *pg_basebackup* you should add at least two more senders to allow for its use.

Like *max_logical_slots*, *max_wal_senders* entries don't cost a large amount of memory, so you can overestimate fairly safely.

max_wal_senders is documented in http://www.postgresql.org/docs/current/static/runtime-config-replication.html the main PostgreSQL documentation.

4.1.1.6.8. wal_keep_segments

Like *max_wal_senders*, the *wal_keep_segments* parameter isn't directly changed by logical replication but is still important for upstream masters. It is not required on downstream-only masters.

wal_keep_segments should be set to a value that allows for some downtime or unreachable periods for downstream masters and for heavy bursts of write activity on the upstream master.

Keep in mind that enough disk space must be available for the WAL segments, each of which is 16MB. If you run out of disk space the server will halt until disk space is freed and it may be quite difficult to free space when you can no longer start the server.

If you exceed the required *wal_keep_segments* and "Insufficient WAL segments retained" error will be reported. See Section 4.1.1.11.

wal_keep_segments is documented in the http://www.postgresql.org/docs/current/static/runtime-config-replication.html the main PostgreSQL manual.

4.1.1.6.9. track_commit_timestamp

Setting this parameter to "on" enables commit timestamp tracking, which is used to implement last-UPDATE-wins conflict resolution. It is also required for use of the *pg_get_transaction_committime* function.

4.1.1.7. Function Reference

The LLSR/BDR patches add several functions to the PostgreSQL core.

4.1.1.7.1. pg_get_transaction_committime

pg_get_transaction_committime(txid integer): Get the timestamp at which the specified transaction, as identified by transaction ID, committed. This function can be useful when monitoring replication lag.

4.1.1.7.2. init_logical_replication

init_logical_replication(slotname name, plugin name, OUT slotname text, OUT xlog_position text) performs the same work as *pg_receivellog --init*, creating a new logical replication slot. Given the slot name to create and the plugin to use for the slot, it returns the fully qualified resulting slot name and the start position in the transaction logs.

This function is mostly useful for implementers of replication tools based on logical replication's features, and is not typically directly useful to end users. Starting a downstream master will automatically create a slot so you do not generally need this function when you are using BDR.

4.1.1.7.3. stop_logical_replication

stop_logical_replication(slotname name) stops logical replication for the named slot, removing its entry from the upstream master. It has the same effect as *pg_receivellog --stop*, but is callable from SQL.

This function is mainly useful for removing unused slots.

4.1.1.7.4. pg_stat_bdr

This function is used by the *pg_stat_bdr* view. You don't need to call it directly.

4.1.1.7.5. bdr_sequence_alloc, bdr_sequence_options, bdr_sequence_setval

These functions are internal to BDR's distributed sequence implementation and are invoked via the sequence access methods. You use ordinary sequence manipulation functions like *nextval* and *setval* to manage distributed sequences, see Section 4.1.1.9.

4.1.1.7.6. pg_xlog_wait_remote_apply

The *pg_xlog_wait_remote_apply(lsn text, pid integer)* function allows you to wait on an upstream master until all downstream masters' replication has caught up to a certain point.

The *lsn* argument is a Log Sequence Number, an identifier for the WAL (Write-Ahead Log) record you want to make sure has been applied on all nodes. The most useful record you will want to wait for is *pg_current_xlog_location()*, as discussed in http://www.postgresql.org/docs/current/static/functions-admin.html PostgreSQL Admin Functions in the manual.

The *pid* argument specifies the process ID of a walsender to wait for. It may be set to zero to wait until the receivers associated with all walsenders on this upstream master have caught up to the specified *lsn*, or to a process ID obtained from *pg_stat_replication.pid* to wait for just one downstream to catch up.

The most common use is: *select pg_xlog_wait_remote_apply(pg_current_xlog_location(), 0)*, which will wait until all downstream masters have applied changes up to the time on the upstream master at which *pg_xlog_wait_remote_apply* was called.

pg_current_xlog_location is not transactional, so unlike things like *current_timestamp* it'll always return the very latest status server-wide, irrespectively of how long the current transaction has been running for and when it started.

4.1.1.7.7. pg_xlog_wait_remote_receive

pg_xlog_wait_remote_receive is the same as *pg_xlog_wait_remote_apply*, except that it only waits until the remote node has confirmed that it's received the given LSN, not until it has actually applied it after receiving it.

4.1.1.8. Catalog Changes

4.1.1.8.1. pg_seqam

To support Section 4.1.1.9, BDR adds an access method abstraction for sequences. It serves a similar purpose to index access methods - it abstracts the implementation of sequence storage from usage of sequences, so the client doesn't need to care whether it's using a distributed sequence, a local sequence, or something else entirely.

This access method is described by the *pg_seqam* table. Two entries are defined as shown in the following listing.

```
postgres=# select * from pg_seqam ;

  seqamname |      seqamalloc      |      seqamsetval      |      seqamoptions

 -----------+----------------------+-----------------------+-----------------------

  local     | sequence_local_alloc | sequence_local_setval | sequence_local_options

  bdr       | bdr_sequence_alloc   | bdr_sequence_setval   | bdr_sequence_options

 (2 rows)
```

local is the traditional local-only sequence access method.

Bdr is for distributed sequences. For more information, see Section 4.1.1.9.

4.1.1.9. Distributed Sequences

Distributed sequences, or global sequences, are a sequence that is synchronized across all the nodes in a BDR cohort. A distributed sequence is more expensive to access than a purely local sequence, but it produces values that are guaranteed unique across the entire cohort.

Using distributed sequences allows you to avoid the problems with inserts conflicts. If you define a *PRIMARY KEY* or *UNIQUE* column with a *DEFAULT nextval(...)* expression that refers to a global sequence shared across all nodes in a BDR cohort, it is not possible for any node to ever get the same value as any other node. When BDR synchronizes inserts between the nodes, they can never conflict.

There is no need to use a distributed sequence if:

- You are ensuring global uniqueness using another method such as:
    - Local sequences with an offset and increment;
    - UUIDs;
    - An externally co-ordinated natural key
- You are using the data in a *TEMPORARY* or *UNLOGGED* table, as these are never visible outside the current node.

You can get a listing of distributed sequences defined in a database with the command shown in the following listing.

```
SELECT *

 FROM pg_class
```

```
 INNER JOIN pg_seqam ON (pg_class.relam = pg_seqam.oid)

 WHERE pg_seqam.seqamname = 'bdr' AND relkind = 'S';
```

See Section 4.1.1.8.1 for information on the new *pg_seqam* catalog table.

New distributed sequences may be created with the *USING* clause to *CREATE SEQUENCE* as shown in the following listing.

```
CREATE SEQUENCE test_seq USING bdr;
```

Once you've created a distributed sequence you may use it with *nextval* like any other sequence. A few limitations and caveats apply to global sequences at time of writing:

- Only an *INCREMENT* of 1 (the default) is supported. Client applications that expect a different increment must be configured to handle increment 1. An extended variant of <tt>nextval</tt> that takes the number of values to obtain as an argument and returns a set of values is planned as an extension to aid in porting.
- *MINVALUE* and *MAXVALUE* are locked at their defaults and may not be changed.
- *START WITH* may not be specified; however, *setval* may be used to set the start value after the sequence is created.
- The *CACHE* directive is not supported.
- Sequence values are handed out in chunks, so if three different nodes all call *nextval* at the same time they might get values 50, 150 and 250. Thus, at time 't' *nextval* on one node may return a value higher than a *nextval* call at time 't+1' on another node. Within a single node the usual rules for *nextval* still apply.

The details used by BDR to manage global sequences are in the *bdr_sequence_values*, *bdr_sequence_elections* and *bdr_votes* tables in the *public* schema, though these details are subject to change.

4.1.1.10. Configuration

Details on individual parameters are described in Section 4.1.1.6.

The following configuration is an example of a simple one-way LLSR replication setup - a single upstream master to a single downstream master.

The upstream master (sender)'s *postgresql.conf* should contain settings like:

```
wal_level = 'logical'       # Include enough info for logical replication

  max_logical_slots = X       # Number of LLSR senders + any receivers

  max_wal_senders = Y         # Y = max_logical_slots plus any physical

                              # streaming requirements

  wal_keep_segments = 5000    # Master must retain enough WAL segments to let

                              # replicas catch up. Correct value depends on

                              # rate of writes on master, max replica downtime

                              # allowable. 5000 segments requires 78GB

                              # in pg_xlog

  track_commit_timestamp = on # Not strictly required for LLSR, only for BDR

                              # conflict resolution.

wal_level = 'logical'       # Include enough info for logical replication

  max_logical_slots = X       # Number of LLSR senders + any receivers

  max_wal_senders = Y         # Y = max_logical_slots plus any physical

                              # streaming requirements

  wal_keep_segments = 5000    # Master must retain enough WAL segments to let
```

```
                              # replicas catch up. Correct value depends on
                              # rate of writes on master, max replica downtime
                              # allowable. 5000 segments requires 78GB
                              # in pg_xlog
  track_commit_timestamp = on # Not strictly required for LLSR, only for BDR
                              # conflict resolution.
```

Downstream (receiver) *postgresql.conf* is shown in the following listing:

```
shared_preload_libraries = 'bdr'


  bdr.connections="name_of_upstream_master"      # list of upstream master nodenames
  bdr.<nodename>.dsn = 'dbname=postgres'         # connection string for connection
                                                 # from downstream to upstream master
  bdr.<nodename>.local_dbname = 'xxx'            # optional parameter to cover the case
                                                 # where the databasename on upstream
                                                 # and downstream master differ.
                                                 # (Not yet implemented)
  bdr.<nodename>.apply_delay                     # optional parameter to delay apply of
                                                 # transactions, time in milliseconds
  bdr.synchronous_commit = off;                  # optional parameter to set the
                                                 # synchronous_commit parameter the
                                                 # apply processes will be using.
                                                 # Safe to set to 'off' unless you're
                                                 # doing synchronous replication.
  max_logical_slots = X                          # set to the number of remotes
```

Note that a server can be both sender and receiver, either two servers to each other or more complex configurations like replication chains/trees.

The upstream (sender) *pg_hba.conf* must be configured to allow the downstream master to connect for replication. Otherwise you'll see errors like the following on the downstream master:

```
FATAL:  could not connect to the primary server: FATAL:  no pg_hba.conf entry for replication connection
from host "[local]", user "postgres"
```

A suitable *pg_hba.conf* entry for a replication connection from the replica server 10.1.4.8 might be:

```
  host    replication     postgres        10.1.4.8/32            trust
```

The user name should match the user name configured in the downstream master's dsn. MD5 password authentication is supported.

For more details on these parameters, see Section 4.1.1.6.

4.1.1.11. Troubleshooting

4.1.1.11.1. Could not access file "bdr": No such file or directory

If you see the error:

```
FATAL:  could not access file "bdr": No such file or directory
```

when starting a database set up to receive BDR replication, you probably forgot to install *contrib/bdr*. See above.

4.1.1.11.2. Invalid value for parameter

An error like:

```
LOG:  invalid value for parameter ...
```

when setting one of these parameters means your server doesn't support logical replication and will need to be patched or updated.

4.1.1.11.3. Insufficient WAL segments retained ("requested WAL segment ... has already been removed")

If *wal_keep_segments* is insufficient to meet the requirements of a replica that has fallen far behind, the master will report errors like:

```
ERROR:  requested WAL segment 0000000100000000000000002D has already been removed
```

Currently the replica errors look like in the following listing, but a more explicit error message for this condition is planned.

```
WARNING:  Starting logical replication

 LOG:  data stream ended

 LOG:  worker process: master (PID 23812) exited with exit code 0

 LOG:  starting background worker process "master"

 LOG:  master initialized on master, remote dbname=master port=5434 replication=true
fallback_application_name=bdr

 LOG:  local sysid 5873181566046043070, remote: 5873181102189050714

 LOG:  found valid replication identifier 1

 LOG:  starting up replication at 1 from 1/2D9CA220
```

The only way to recover from this fault is to re-seed the replica database.

This fault could be prevented with feedback from the replica to the master, but this feature is not planned for the first release of BDR. Another alternative considered for future releases is making wal_keep_segments a dynamic parameter that is sized on demand.

Monitoring of maximum replica lag and appropriate adjustment of wal_keep_segments will prevent this fault from arising.

4.1.1.11.4. Couldn't find logical slot

An error like:

```
ERROR:  couldn't find logical slot "bdr: 16384:5873181566046043070-1-24596:"
```

on the upstream master suggests that a downstream master is trying to connect to a logical replication slot that no longer exists. The slot cannot be re-created, so it is necessary to re-seed the downstream replica database.

4.1.1.12. Operational Issues and Debugging

In LLSR there are no user-level (i.e. SQL visible) ERRORs that have special meaning. Any ERRORs generated are likely to be serious problems of some kind, apart from apply deadlocks, which are automatically re-tried.

4.1.1.13. Monitoring

The following views are available for monitoring replication activity:

- http://www.postgresql.org/docs/current/static/monitoring-stats.html#MONITORING-STATS-VIEWS-TABLE pg_stat_replication
- *pg_stat_logical_decoding* (described below)
- *pg_stat_bdr* (described below)

The following configuration and logging parameters are useful for monitoring replication:

- http://www.postgresql.org/docs/current/static/runtime-config-logging.html#GUC-LOG-LOCK-WAITS log_lock_waits

4.1.1.13.1. pg_stat_logical_decoding

The new *pg_stat_logical_decoding* view is specific to logical replication. It is based on the underlying *pg_stat_get_logical_replication_slots* function and has the following structure:

```
 View "pg_catalog.pg_stat_logical_decoding"

          Column          |  Type   | Modifiers

--------------------------+---------+-----------

 slot_name                | text    |

 plugin                   | text    |

 database                 | oid     |

 active                   | boolean |

 xmin                     | xid     |

 last_required_checkpoint | text    |
```

It contains one row for every connection from a downstream master to the server being queried (the upstream master). On a standalone PostgreSQL server or a downstream-only master this view will contain no rows.

- *slot_name*: An internal name for a given logical replication slot (a connection from a downstream master to this upstream master). This slot name is used by the downstream master to uniquely identify its self and is used with the *pg_receivellog* command when managing logical replication slots. The slot name is composed of the decoding plugin name, the upstream database oid, the downstream system identifier (from *pg_control*), the downstream slot number, and the downstream database oid.
- *plugin*: The logical replication plugin being used to decode WAL archives. You'll generally only see *bdr_output* here.
- *database*: The oid of the database being replicated by this slot. You can get the database name by joining on *pg_database.oid*.
- *active*: Whether this slot currently has an active connection.
- *xmin*: The lowest upstream master transaction ID this replication slot can "see", like the xmin of a transaction or prepared transaction. xmin should keep on advancing as replication continues. If xmin stops advancing that's a sign that replication has stalled, possibly leading to the accumulation of bloat in the system catalogs. You can turn this txid into the time it was committed with *pg_get_transaction_committime* (see Section 4.1.1.7.1) for monitoring purposes.
- *last_required_checkpoint*: The checkpoint identifying the oldest WAL record required to bring this slot up to date with the upstream master. (This column is likely to be removed in a future version).

4.1.1.13.2. pg_stat_bdr

The *pg_stat_bdr* view is supplied by the *bdr* extension. It provides information on a server's connection(s) to its upstream master(s). It is not present on upstream-only masters.

The primary purpose of this view is to report statistics on the progress of LLSR apply on a per-upstream master connection basis.

```
View structure:


         View "public.pg_stat_bdr"

        Column       |  Type  | Modifiers

 --------------------+--------+-----------

  rep_node_id        | oid    |

  riremotesysid      | name   |

  riremotedb         | oid    |

  rilocaldb          | oid    |

  nr_commit          | bigint |

  nr_rollback        | bigint |

  nr_insert          | bigint |

  nr_insert_conflict | bigint |

  nr_update          | bigint |

  nr_update_conflict | bigint |

  nr_delete          | bigint |

  nr_delete_conflict | bigint |

  nr_disconnect      | bigint |
```

Fields:

- *rep_node_id*: An internal identifier for the replication slot.
- *riremotesysid*: The remote database system identifier, as reported by the *Database system identifier* line of *pg_controldata /path/to/datadir*
- *riremotedb*: The remote database OID, i.e. the *oid* column of the remote server's *pg_catalog.pg_database* entry for the replicated database. You can get the database name with *select datname from pg_database where oid = 12345* (where '12345' is the *riremotedb* oid).
- *rilocaldb*: The local database OID, with the same meaning as *riremotedb* but with oids from the local system.

''The rest of the rows are statistics about this upstream master slot'':

- *nr_commit*: Number of commits applied to date from this master
- *nr_rollback*: Number of rollbacks performed by this apply process due to recoverable errors (deadlock retries, lost races, etc) or unrecoverable errors like mismatched constraint errors.
- *nr_insert*: Number of *INSERT*s performed
- *nr_insert_conflict*: Number of *INSERT*s that resulted in conflicts.
- *nr_update*: Number of *UPDATE*s performed
- *nr_update_conflict*: Number of *UPDATE*s that resulted in conflicts.

- nr_delete: Number of deletes performed
- *nr_delete_conflict*: Number of deletes that resulted in conflicts.
- *nr_disconnect*: Number of times this apply process has lost its connection to the upstream master since it was started.

This view does not contain any information about how far behind the upstream master this downstream master is. The upstream master's *pg_stat_logical_decoding* and *pg_stat_replication* views must be queried to determine replication lag.

4.1.1.13.3. Monitoring Replication Status and Lag

As with any replication setup, it is vital to monitor replication status on all BDR nodes to ensure no node is lagging severely behind the others or is stuck.

In the case of BDR a stuck or crashed node will eventually cause disk space and table bloat problems on other masters so stuck nodes should be detected and removed or repaired in a reasonably timely manner. Exactly how urgent this is depends on the workload of the BDR group.

The *pg_stat_logical_decoding* view described above may be used to verify that a downstream master is connected to its upstream master by querying it on the upstream side - the *active* boolean column is *t* if there's a downstream master connected to this upstream.

The *xmin* column provides an indication of whether replication is advancing; it should increase as replication progresses. You can turn this into the time the transaction was committed on the master by running *pg_get_transaction_committime(xmin)* ''on the upstream master''. Since txids are different between upstream and downstream masters, running it on a downstream master with a txid from the upstream master as input would result in an error and/or incorrect result. An example is shown in the following listing.

```
postgres=# select slot_name, plugin, database, active, xmin,

             pg_get_transaction_committime(xmin)

             FROM pg_stat_logical_decoding ;

 -[ RECORD 1 ]-----------------+---------------------------------------

 slot_name                     | bdr: 12910:5882534759278050995-1-12910:

 plugin                        | bdr_output

 database                      | 12910

 active                        | f

 xmin                          | 1827

 pg_get_transaction_committime | 2013-05-27 06:14:36.851423+00
```

4.1.1.14. Table and Index Usage Statistics

Statistics on table and index usage are updated normally by the downstream master. This is essential for correct function of auto-vacuum. If there are no local writes on the downstream master and stats have not been reset these two views should show matching results between upstream and downstream:

- *pg_stat_user_tables*
- *pg_statio_user_tables*

Since indexes are used to apply changes, the identifying indexes on downstream side may appear more heavily used with workloads that perform *UPDATE*s and DELETEs than non-identifying indexes are.

The built-in index monitoring views are:

- *pg_stat_user_indexes*
- pg_statio_user_indexes

All these views are discussed in http://www.postgresql.org/docs/current/static/monitoring-stats.html#MONITORING-STATS-VIEWS-TABLE the PostgreSQL documentation on the statistics views.

4.1.1.15. Starting, Stopping, and Managing Replication

Replication is managed with the *postgresql.conf* settings described in "Parameter Reference" and "Configuration" above, and using the *pg_receivellog* utility command.

4.1.1.15.1. Starting a new LLSR Connection

Logical replication is started automatically when a database is configured as a downstream master in *postgresql.conf* (see Section 4.1.1.10) and the postmaster is started. No explicit action is required to start replication, but replication will not actually work unless the upstream and downstream databases are identical within the requirements set by LLSR in Section 4.1.1.3.

pg_dump and *pg_restore* may be used to set up the new replica's database.

4.1.1.15.2. Viewing Logical Replication Slots

Examining the state of logical replication is discussed in Section 4.1.1.13.

4.1.1.15.3. Temporarily Stopping an LLSR Replica

LLSR replicas can be temporarily stopped by shutting down the downstream master's postmaster.

If the replica is not started back up before the upstream master discards the oldest WAL segment required for the downstream master to resume replay, as identified by the *last_required_checkpoint* column of *pg_catalog.pg_stat_logical_decoding* then the replica will not resume replay. The error from Section 4.1.1.11.3 will be reported in the upstream master's logs. The replica must be re-created for replication to continue.

4.1.1.15.4. Removing an LLSR Replica Permanently

To remove a replication connection permanently, remove its entries from the downstream master's *postgresql.conf*, restart the downstream master, then use *pg_receivellog* to remove the replication slot on the upstream master.

It is important to remove the replication slot from the upstream master(s) to prevent xid wrap-around problems and issues with table bloat caused by delayed vacuum.

4.1.1.15.5. Cleaning up Abandoned Replication Slots

To remove a replication slot that was used for a now-defunct replica, find its slot name from the Section 4.1.1.13.1 view on the upstream master then run:

```
pg_receivellog -p 5434 -h master-hostname -d dbname \

    --slot='bdr: 16384:5873181566046043070-1-16384:' --stop
```

where the argument to '--slot' is the slot name you found from the view.

You may need to do this if you've created and then deleted several replicas so *max_logical_slots* has filled up with entries for replicas that no longer exist.

4.1.2. Bi-Directional Replication

Bi-Directional replication is built directly on LLSR by configuring two or more servers as both upstream "and" downstream masters of each other.

All of the Log Level Streaming Replication documentation applies to BDR and should be read before moving on to reading about and configuring BDR.

4.1.2.1. Bi-Directional Replication Use Cases

Bi-Directional Replication is designed to allow a very wide range of server connection topologies. The simplest to understand would be two servers each sending their changes to the other, which would be produced by making each server the downstream master of the other and so using two connections for each database.

Logical and physical streaming replication are designed to work side-by-side. This means that a master can be replicating using physical streaming replication to a local standby server, while at the same time replicating logical changes to a remote downstream master. Logical replication works alongside cascading replication also, so a physical standby can feed changes to a downstream master, allowing upstream master sending to physical standby sending to downstream master.

4.1.2.1.1. Simple Multi-Master Pair

A simple mulit-master "HA Cluster" with two servers:

- Server "Alpha" – Master
- Server "Bravo" – Master

Configuration

Alpha:

```
wal_level = 'logical'

max_logical_slots = 3

max_wal_senders = 4

wal_keep_segments = 5000

shared_preload_libraries = 'bdr'

bdr.connections="bravo"

bdr.bravo.dsn = 'dbname=dbtoreplicate'

track_commit_timestamp = on
```

Bravo:

```
wal_level = 'logical'

max_logical_slots = 3

max_wal_senders = 4

wal_keep_segments = 5000

shared_preload_libraries = 'bdr'

bdr.connections="alpha"

bdr.alpha.dsn = 'dbname=dbtoreplicate'

track_commit_timestamp = on
```

See Section 4.1.1.10 for an explanation of these parameters.

4.1.2.1.2. HA and Logical Standby

Downstream masters allow users to create temporary tables, so they can be used as reporting servers.

"HA Cluster":

- Server "Alpha" - Current Master
- Server "Bravo" - Physical Standby - unused, apart from as failover target for Alpha - potentially specified in synchronous_standby_names
- Server "Charlie" - "Logical Standby" - downstream master

4.1.2.1.3. Very High Availability Multi-Master

A typical configuration for remote multi-master would then be:

- Site 1
    - Server "Alpha" - Master - feeds changes to Bravo using physical streaming with sync replication
    - Server "Bravo" - Physical Standby - feeds changes to Charlie using logical streaming
- Site 2
    - Server "Charlie" - Master - feeds changes to Delta using physical streaming with sync replication
    - Server "Delta" - Physical Standby - feeds changes to Alpha using logical streaming

Bandwidth between Site 1 and Site 2 is minimised.

4.1.2.1.4. 3-Remote Site Simple Multi-Master Plex

BDR supports "all to all" connections, so the latency for any change being applied on other masters is minimised. (Note that early designs of multi-master were arranged for circular replication, which has latency issues with larger numbers of nodes)

- Site 1
    - Server "Alpha" - Master - feeds changes to Charlie, Echo using logical streaming
- Site 2
    - Server "Charlie" - Master - feeds changes to Alpha, Echo using logical streaming replication
- Site 3
    - Server "Echo" - Master - feeds changes to Alpha, Charlie using logical streaming replication

Configuration

If you wanted to test this configuration locally you could run three PostgreSQL instances on different ports. Such a configuration would look like the following if the port numbers were used as node names for the sake of notational clarity:

```
Config for node_5440:



 port = 5440

 bdr.connections='node_5441,node_5442'
```

```
 bdr.node_5441.dsn='port=5441 dbname=postgres'

 bdr.node_5442.dsn='port=5442 dbname=postgres'


Config for node_5441:


 port = 5441

 bdr.connections='node_5440,node_5442'

 bdr.node_5440.dsn='port=5440 dbname=postgres'

 bdr.node_5442.dsn='port=5442 dbname=postgres'


Config for node_5442:


 port = 5442

 bdr.connections='node_5440,node_5441'

 bdr.node_5440.dsn='port=5440 dbname=postgres'

 bdr.node_5441.dsn='port=5441 dbname=postgres'
```

In a typical real-world configuration each server would be on the same port on a different host instead.

4.1.2.1.5. 3-Remote Site Simple Multi-Master Circular Replication

Simpler config uses "circular replication". This is simpler but results in higher latency for changes as the number of nodes increases. It's also less resilient to network disruptions and node faults.

- Site 1
    - Server "Alpha" - Master - feeds changes to Charlie using logical streaming replication
- Site 2
    - Server "Charlie" - Master - feeds changes to Echo using logical streaming replication
- Site 3
    - Server "Echo" - Master - feeds changes to Alpha using logical streaming replication

Regrettably this doesn't actually work yet because we don't cascade logical changes (yet).

Configuration

Using node names that match port numbers, for clarity

```
Config for node_5440:


 port = 5440

 bdr.connections='node_5441'

 bdr.node_5441.dsn='port=5441 dbname=postgres'
```

```
Config for node_5441:


 port = 5441

 bdr.connections='node_5442'

 bdr.node_5442.dsn='port=5442 dbname=postgres'


Config for node_5442:


 port = 5442

 bdr.connections='node_5440'

 bdr.node_5440.dsn='port=5440 dbname=postgres'
```

This would usually be done in the real world with databases on different hosts, all running on the same port.

4.1.2.1.6. 3-Remote Site Max Availability Multi-Master Plex

- Site 1
  - Server "Alpha" - Master - feeds changes to Bravo using physical streaming with sync replication
  - Server "Bravo" - Physical Standby - feeds changes to Charlie, Echo using logical streaming
- Site 2
  - Server "Charlie" - Master - feeds changes to Delta using physical streaming with sync replication
  - Server "Delta" - Physical Standby - feeds changes to Alpha, Echo using logical streaming
- Site 3
  - Server "Echo" - Master - feeds changes to Foxtrot using physical streaming with sync replication
  - Server "Foxtrot" - Physical Standby - feeds changes to Alpha, Charlie using logical streaming

Bandwidth and latency between sites is minimised.

Here the config is left as an exercise for the reader.

4.1.2.1.7. N-site Symmetric Cluster Replication

Symmetric cluster is where all masters are connected to each other.

N=19 has been tested and works fine.

N masters requires N-1 connections to other masters, so practical limits are <100 servers, or less if you have many separate databases.

The amount of work caused by each change is O(N), so there is a much lower practical limit based upon resource limits. A future option to limit to filter rows/tables for replication becomes essential with larger or more heavily updated databases, which is planned.

4.1.2.2. Conflict Avoidance

4.1.2.2.1. Distributed Locking

Some clustering systems use distributed lock mechanisms to prevent concurrent access to data. These can perform reasonably when servers are very close but cannot support geographically distributed applications as very low latency is critical for acceptable performance.

Distributed locking is essentially a pessimistic approach, whereas BDR advocates an optimistic approach: avoid conflicts where possible but allow some types of conflict to occur and and resolve them when they arise.

4.1.2.2.2. Global Sequences

Many applications require unique values be assigned to database entries. Some applications use GUIDs generated by external programs, some use database-supplied values. This is important with optimistic conflict resolution schemes because uniqueness violations are "divergent errors" and are not easily resolvable.

The SQL standard requires Sequence objects which provide unique values, though these are isolated to a single node. These can then be used to supply default values using *DEFAULT nextval('mysequence')*, as with PostgreSQL's *SERIAL* pseudo-type.

BDR requires sequences to work together across multiple nodes. This is implemented as a new *SequenceAccessMethod* API (SeqAM), which allows plugins that provide get/set functions for sequences. Global Sequences are then implemented as a plugin which implements the SeqAM API and communicates across nodes to allow new ranges of values to be stored for each sequence.

4.1.2.3. Conflict Detection & Resolution

Because local writes can occur on a master, conflict detection and avoidance is a concern for basic LLSR setups as well as full BDR configurations.

4.1.2.3.1. Lock Conflicts

Changes from the upstream master are applied on the downstream master by a single apply process. That process needs to RowExclusiveLock on the changing table and be able to write lock the changing tuple(s). Concurrent activity will prevent those changes from being immediately applied because of lock waits. Use the http://www.postgresql.org/docs/current/static/runtime-config-logging.html#GUC-LOG-LOCK-WAITS log_lock_waits facility to look for issues with apply blocking on locks.

By concurrent activity on a row, we include

explicit row level locking (*SELECT ... FOR UPDATE/FOR SHARE*)

- locking from foreign keys
- implicit locking because of row *UPDATE*s, INSERTs or DELETEs, either from local activity or apply from other servers

4.1.2.3.2. Data Conflicts

Concurrent updates and deletes may also cause data-level conflicts to occur, which then require conflict resolution. It is important that these conflicts are resolved in a consistent and idempotent manner so that all servers end up with identical results.

Concurrent updates are resolved using last-update-wins strategy using timestamps. Should timestamps be identical, the tie is broken using system identifier from *pg_control* though this may change in a future release.

*UPDATE*s and *INSERT*s may cause uniqueness violation errors because of primary keys, unique indexes and exclusion constraints when changes are applied at remote nodes. These

are not easily resolvable and represent severe application errors that cause the database contents of multiple servers to diverge from each other. Hence these are known as "divergent conflicts". Currently, replication stops should a divergent conflict occur. The errors causing the conflict can be seen in the error log of the downstream master with the problem.

Updates which cannot locate a row are presumed to be *DELETE UPDATE* conflicts. These are accepted as successful operations but in the case of *UPDATE* the data in the UPDATE is discarded.

All conflicts are resolved at row level. Concurrent updates that touch completely separate columns can result in "false conflicts", where there is conflict in terms of the data, just in terms of the row update. Such conflicts will result in just one of those changes being made, the other discarded according to last update wins. It is not practical to decide when a row should be merged and when a last-update-wins strategy should be used at the database level; such decision making would require support for application-specific conflict resolution plugins.

Changing unlogged and logged tables in the same transaction can result in apparently strange outcomes since the unlogged tables aren't replicated.

Examples

As an example, let's say we have two tables Activity and Customer. There is a Foreign Key from Activity to Customer, constraining us to only record activity rows that have a matching customer row. We update a row on Customer table on NodeA. The change from NodeA is applied to NodeB just as we are inserting an activity on NodeB. The inserted activity causes a FK check.

4.2. CumuloNimbo

In what follows we provide the user guide for the deployment and execution of the CumuloNimbo ultra-scalable database in the 4CaaSt platform.

The CumuloNimbo ultra-scalable database is deployed and managed through Chef cookbooks designed following the 4CaaSt template. There are 5 cookbooks, each of them deploying a different subsystem or subset of subsystems:

- *cn_hdfs_all*: Deploys and starts HDFS
- *cn_hbase_all*: Deploys and starts HBase with Zookeeper running locally
- *cn_tm_all*: Deploys and starts the Transaction Manager
- *cn_qe_all*: Deploys and starts Derby
- *cn_tomcat_all*: Deploys and starts Tomcat

The cookbooks follow a common scheme. They all create these directories:

- Software: */home/<user>/<app>*
- Data: */local/<user>/<app>*
- Java: */home/<user>/jdk1.6.0_22_x32*

Where:

- <user> is declared with the attribute *user*.
- <app> depends on the cookbook being installed:
    - cn_hdfs_all: HDFS-SHBase-0.92
    - cn_hbase_all: SHBase-0.92
    - cn_tm_all: txnmgmt
    - cn_qe_all: derbyBIN on home and derbyDATA on local
    - cn_tomcat_all: apache-tomcat-6.0.37

The cookbooks should be executed in the following order:

1. cn_hdfs-all
2. cn_hbase-all
3. cn_txnmgmt-all
4. cn_qe-all
5. cn_tomcat-all

For each cookbook the two recipes required by 4CaaSt, deploy and start, should be executed in the following order:

1. Deploy_PIC.rb
2. Start_PIC.rb

The cookbooks have multiple attributes that are described below.
The default values of attributes are provided in:

- default.rb

The attributes file provides all the customizable parameters although most of them can be left as they are. The only exception is the attribute *user* which must be an existing linux user:

user is the owner of the files and directories under which all the processes will run. Default *user* is *ubuntu*.

4.3. Java Open Application Server – JOnAS

JOnAS documentation is available online at: http://jonas.ow2.org/JONAS_5_3_0_M7/doc/doc-en/html/. To support JOnAS accounting, the WP5 JASMINe Probe monitoring source component is used, the way it is installed is described in D5.3.3 [12].

4.4. Performance Isolation Benchmarking

In this section we provide the information for using the performance isolation measurement framework.

4.4.1. Metrics

Actually the Metrics are rather a scientific contribution. Nevertheless, for applying them in real scenarios one has to carefully define the reference workload and QoS Metrics which has a major impact onto the later results. A user of these metrics should be aware this fact. We recommend the following:

The QoS metric we focus on is usually the response time as here an increasing value means a negative performance as this was requested by the metrics algorithms. The time is measured from the moment a request leaves a tenant to the point in time a tenant receives the response. As a measure for the workload caused by the tenants we recommend the number of users associated with each tenant. In a MTA the workload induced by the tenants is rather homogeneous (except the amount). Thus all users send requests of the same type which makes different workload configurations comparable and allows to assume a similar affect onto the system at increasing loads independent from the tenant sending the requests. We expect that the system runs with a high utilization for economic reasons in reality, thus we should use a high utilization for the reference workload. Another reason for running under high utilization is our goal of evaluating performance isolation aspects. In a system with low utilization, the increased workload of one tenant would have low impact. Therefore, we should measure the systems overall throughput with increasing number of users. The point at which the system has the highest throughput could be used as reference workload. If this point is already exceeding the SLAs the workload at which the highest throughput with still being within the SLAs should be selected as reference workload.

4.4.2. Measurement Framework

For some general information concerning the SoPeCo and how to use it we refer to http://www.sopeco.org . In the following we provide some detailed information on how to implement a Runtime Environment Connector for a specific system and how to setup a configuration for a concrete measurement.

4.4.2.1. Creation of an Isolation Specific Measurement Environment Connector

To implement a Measurement Environment Controller one has to realize the Abstract Class *MultiTenantMeasurementEnvironmentController* provided in the jar file at https://svn.forge.morfeo-project.org/4caast/trunk/WP7/release3.0/isolation. Similar to the original version *MeasurementEnvironmentController* it provides three abstract methods.

- *initialize()* initializes the target system with the given parameters from the Setup of the Measurement configuration.
- *prepareExperimentSeries*() prepares the measurement environment with given parameter values. It may run some test experiments on the target system. However, the results from the test experiments will not be stored.
- The *runExperiment*() method is the main method for running an experiment on the target system. The results from the experiment are stored in the Persistence.

The parameters transferred are key-value pairs. Especially for the initialize and prepareExperimentSeries these values are very specific to the concrete implementation and being set by the user in the SoPeCo configuration for the environment. The runExperiment method gets a TenantCollection transferred which contains several Tenants. Each of these Tenants carry the following information: tenant_id, tenant_characteristic (abiding, disruptive), workload and targetIdentifier (usually referring to the hostname). Furthermore the field response time is used to return the observed average response time to the SoPeCo.

4.4.2.2. Setup of the Measurement Strategy

In the Web Version of the SoPeCo we currently only support the Impact Based metrics. Select the Isolation Measurement Strategy from the available ones and provide the following information for the measurement strategy, see Table 1.

Parameter	**Description**	**Example**
Abiding Tenant host	The hostnames of the abiding tenants separated by commas. We assume different hostnames for each tenant.	Hostname-1,hostname-2
Disruptive Tenant host	The hostnames of the disruptive tenant.	Hostname-3
Abiding Tenant port	The ports for the abiding tenants.	8080,8080
Disruptive Tenant port	The ports for the disruptive tenant.	8080
Abiding workload	The amount of workload for each abiding tenant separated by commas.	1500,1200
Disruptive workload	Describes the range of disruptive workloads which should be used for the measurements. Required information is a *Start* value an *End* value and a *Step* width.	Start=1200 End=5000 Step=200

Duration	Describes the duration of how long one particular workload configuration should be measured in minutes.	10

Table 1. Overview of Parameters for Measurement Strategy

4.4.3. TPC-W

Once the MT-TPCW is deployed the primary user interface is the client console. In Table 2 we present an overview of the commands supported by the console.

Command	**Meaning**
create	Creates a new Load Driver instance for one single tenant. It starts an interactive mode to ask for the needed information. To use default values press enter without any value. Mandatory fields are the number of Emulated Browsers (#Users), the Destination Address (e.g., http://hostname:8080/) and a description identifying the tenant in the client console.
list	Shows a list of all tenants, their id, description and amount of users running.
kill	Stops the load driver generation of workload.
add <tenantId> <amount>	Ads <amount> of users to the specific tenant.
remove <tenantId> <amount>	Removes <amount> of users to the specific tenant.
loggerStart	Starts the response time logger for all tenants.
loggerStop	Stops the response time logger for all tenants.
loggerReset	Resets the response time logger. The logger is still logging. All data logged up to that point is removed.
loggerEvaluate	Returns the average response time for each tenant in the time from the last loggerReset or loggerStart to the current moment in time or the time loggerStop was triggered. loggerEvaluate does not reset the logger.

Table 2. Overview of Commands Supported by the Client Console

4.5. Bonita Open Solution – BOS

There are no changes to be highlighted with respect to user interaction. Thus, the interested reader is referred to D7.3.1 [1] and D7.3.2 [2] for further information.

4.6. Orchestra

Orchestra User Guide is available online http://download.forge.objectweb.org/orchestra/Orchestra-4.9.0-M4-UserGuide.pdf.

4.7. Extension of Apache ServiceMix for Multi-Tenancy

With respect to functionality extension in RP3 we focused on security and extended the multi-tenant HTTP BC of ESB^{MT} in order to enable communication via HTTPS. Apart from this extension, the user manual provided in the previous version of this deliverable is valid and the interested reader is for details referred to [2]. Thus, in the following we emphasize how to extend the test cases in order to use encrypted communication via HTTPS.

For utilizing the whole extended ServiceMix environment please refer to the SoapUI [10] projects provided in:

https://svn.forge.morfeo-project.org/4caast/trunk/WP7/ExtendedApacheServiceMix/release-RP2/miscellaneous/SoapUI_TestSuits

The extension for usage of HTTPS only affects the communication via SOAP over HTTPS. As before the SOAP Binding Component can still be accessed under the following multi-tenant URL using HTTP:

http://**<VM IP Address>**:8193/tenant-services/**<registered tenant URL>/<Service Name>/<Endpoint Name>**/main.wsdl

For example:

http://187.212.86.2:8193/tenant-services/taxicompany.example.org/httpSoapConsumer/TaxiProviderHttpSoapConsumerEndpoint/main.wsdl

In addition, the SOAP Binding Component can now also be accessed under the following multi-tenant URL using HTTPS:

https://**<VM IP Address>**:8193/tenant-services/**<registered tenant URL>/<Service Name>/<Endpoint Name>**/main.wsdl

For example:

https://187.212.86.2:8193/tenant-services/taxicompany.example.org/httpSoapConsumer/TaxiProviderHttpSoapConsumerEndpoint/main.wsdl

5. Conclusion

This deliverable provides the final version of the documentation for the prototypical implementations of the immigrant PaaS technologies extended for cloud-awareness and integrated into the 4CaaSt platform in WP7 during RP3 of the 4CaaSt project. We focus on providing information on the architectures of the prototypes (Section 2), component management including build, installation, setup and configuration (Section 3), and a user guide (Section 4). The specification and design of the prototypes can be found in previous deliverable D7.2.3 [3].

All in all with this deliverable we have reached the WP7 goal to enable cloud-awareness for the building blocks and their seamless collaboration and interaction with other work packages to ensure coverage of the whole building block lifecycle including offering in the 4CaaSt marketplace, dynamic scalability and deployment, monitoring, as well as accounting and billing.

6. References

[1] 4CaaSt Consortium: D7.3.1 Immigrant PaaS Technologies: Experimental Prototype of Software Components and Documentation, Version 1.0, January, 2012.

[2] 4CaaSt Consortium: D7.3.2 Immigrant PaaS Technologies: Experimental Prototype of Software Components and Documentation, Version 1.0, January, 2013.

[3] 4CaaSt Consortium: D7.2.3 Immigrant PaaS Technologies: Components Design and Open Specification, Version 1.0, July, 2013.

[4] Steve Strauch, Vasilios Andrikopoulos, Santiago Gómez Sáez, and Frank Leymann: Implementation and Evaluation of a Multi-Tenant Open-Source ESB. In: Proceedings of ESOCC'12, 2012. (to appear)

[5] Apache Software Foundation: Apache ServiceMix. http://servicemix.apache.org.

[6] 4CaaSt Consortium: D8.1.4 Use Case Applications eMarketPlace for SMEs: Report on Integration, Version 2.0, April 10, 2013.

[7] 4CaaSt Consortium: D8.1.6 Use Case Applications eMarketPlace for SMEs: Report on Experimentation Results, Version 1.0, July 2, 2013.

[8] 4CaaSt Consortium: D2.3.3 Service Engineering and Lifecycle Management: Experimental Prototype of Software Components and Documentation, Version 1.0, August, 2013.

[9] 4CaaSt Consortium: D3.3.3 Marketplace Functions: Experimental Prototype of Software Components and Documentation, Version 1.0, August, 2013.

[10] SmartBear Software. soapUI. http://www.soapui.org.

[11] 4CaaSt Consortium: D5.2.3 Administration, Accounting, Monitoring and Analytics: Components Design and Open Specification, Version 1.0, July, 2013.

[12] 4CaaSt Consortium: D5.3.3 Administration, Accounting, Monitoring and Analytics: Experimental Prototype of Software Components and Documentation, Version 1.0, August, 2013.

All links have been last checked on September 9, 2013

