PADRES Developers Guide

PADRES Development Team

July 16, 2012

1 Introduction

This documentation is targeted at developers who are interested in studying, using and extending PADRES. PADRES is an open source distributed publish/subscribe system that is available for download (see here for more details). Please make sure that you have gone through the *PADRES User Guide* before following this *Developer Guide*. Both guides assume that you have a general understanding of content-based publish/subscribe systems. For further information, please refer the PADRES web site[1].

PADRES code base includes the Java source code of PADRES core components, tools, tests, and demos; scripts and configuration files to run different PADRES components; and data files for included demos. PADRES has two primary components: broker and client. All the PADRES core features are implemented inside the broker, while the client provides a pub/sub interface.

2 Software Requirements

PADRES is developed in Java using the Eclipse[2] IDE. Therefore, many sections in this developer guide are written specific to Eclipse. PADRES was developed using Java 6 and requires the library dependencies specified in Table 1.

External Libraries	Where it is used (package)	What functionality it provides
junit.jar	`padres.test.junit`	JUnit testing
jess.jar v.6	`padres.broker.router.matching.jess`	Jess Matching Engine
pg73jdbc3.jar	`padres.broker.comm.db`	PostgreSQL (Only in Linux)
log4j-1.2.13.jar	everywhere	PADRES Logging
jung-1.7.6.jar	`padres.tools.padresmonitor`	PADRES Monitor GUI
colt.jar	used externally	Monitor GUI
simple-3.1.3.jar	`ca.utoronto.msrg.padres.broker.management.web`	Web management interface

Table 1: External libraries required by PADRES.

3 PADRES Package Structure

PADRES is distributed as an open source project (see here for instructions to retrieve the source). The project's directory structure is as follows:

[1] http://padres.msrg.utoronto.ca

[2] http://www.eclipse.org

```
|-- bin     --> Contains scripts to run different PADRES components.
|-- build   --> Where the compiled PADRES class files are placed.
|-- demo    --> PADRES demo related code, data, configuration files,
|               and run scripts.
|-- doc     --> PADRES JavaDoc.
|-- etc     --> PADRES configuration files (for the core components
|               and demos.)
|-- lib     --> External Java libraries required for PADRES
|-- libdev  --> This directory contains libraries used to build or
|               test the code. These files do not need to be
|               distributed in binary releases.
|-- release --> Contains scripts and other files required to prepare
|               PADRES release.
`-- src     --> PADRES Java codes.
```

PADRES src: PADRES source Java packages are under the `src/` directory and are structured as follows:

```
src/ca/utoronto/msrg/padres/
|-- broker            --> PADRES broker related components.
|   |-- brokercore    --> Core codes of the PADRES broker.
|   |-- controller    --> Codes for PADRES broker controls.
|   |-- management    --> Codes for PADRES broker management.
|   |-- monitor       --> Codes for PADRES broker monitor.
|   |-- router        --> Codes for the router inside the PADRES
|   |                     broker. It includes preprocessing, matching,
|   |                     post processing, and forwarding.
|   `-- webmonitor    --> Web-based monitor for PADRES broker.
|-- client            --> Basic client interface, which can be
|                         extended to produce clients to suite different
|                         applications.
|-- common            --> Classes and methods that use multiple PADRES
|   |                     components (broker, client, tools.)
|   |-- comm          --> Code for the communication layer; includes
|   |                     packages for RMI-based and socket-based
|   |                     communication protocols.
|   |-- message       --> PADRES messages used for communication
|   |                     between PADRES components. Includes message
|   |                     parser to be created using JavaCC.
|   `-- util          --> Utility codes used by various PADRES
|                         components.
|-- test              --> Directory containing codes to test the
|   |                     correctness of PADRES functionalities.
|   `-- junit         --> JUnit test cases to test different
|                         functionalities of PADRES.
`-- tools
    |-- cliclient     --> An interactive pub/sub client that connects
    |                     to a PADRES broker. User interacts with it
    |                     via a command line interface.
```

```
|-- guiclient     --> Similar to the above, but provides a GUI
|                     interface.
|-- padresmonitor --> A GUI PADRES monitor that can visually
|                     present the operation of a PADRES
|                     overlay. It also provides functions to
|                     control/query brokers in the overlay.
|-- panda         --> A CLI-based PADRES deployment tool.
`-- webclient     --> Similar to the cliclient, but provides a web
                      interface.
```

PADRES demos:

```
demo
|-- /src/ca/utoronto/msrg/padres/demo
|   |-- stockquote    --> A simple application to emulate a stock
|   |                     quote market.
|   |-- webclient     --> A web-based client.
|   `-- workflow      --> Publish/subscribe system that supports work
|                         flow application.
|-- bin               --> run scripts for the demos
|-- data              --> data files to feed the demos
`-- etc               --> configuration files to be read by the demos
```

4 Building PADRES

4.1 In Eclipse

Note: Before proceeding, please make sure you have the JavaCC plug-in (available from here) for Eclipse installed.

- Open "Project" → "Properties"
- In "Java Build Path" → "Source" dialog:
  - Make sure <project_dir>/src and <project_dir>/demo/src are set as "Source folders on build path".
  - Make sure <project_dir>/build is set as "Default output folder".
- In "Java Build Path" → "Libraries" dialog:
  - Add the external libraries in <project_dir>/lib using "Add JARs".
  - Note: If any of these external libraries is already installed in your system (and can be accessed by any Java project), you don't have to add the relevant jar file.
- In "JavaCC options" → "JavaCC Options" dialog:
  - Make sure the following boxes are checked:
    - OPTIMIZE_TOKEN_MANAGER (default:true)
    - ERROR_REPORTING (default:true)
    - BUILD_PARSER (default:true)

    - BUILD_TOKEN_MANAGER (default:true)
    - SANITY_CHECK (default:true)
    - KEEP_LINE_COLUMN (default:true)
  - Also make sure that the STATIC (default:true) box is unchecked and the OUTPUT_DIRECTORY is blank.

- Click "Apply" and "OK"

After completing the above steps, use "Project" → "Build Project" to build the project (building the project this way is not necessary if "Build Automatically" is enabled.)

4.2 Using `ant`

From the project directory, use the following at a shell prompt:

```
$ ant clean compile
```

5 Running PADRES

5.1 Running BrokerCore

This section describes how to run a PADRES broker. Consider a broker topology consisting of two brokers (with IDs "BrokerA" and "BrokerB") running on localhost and connected to one another. PADRES supports two types of binding, namely RMI and socket. For this example, the socket protocol is used and BrokerA and BrokerB are listening at the port 3000 and 4000 respectively.

5.1.1 From Eclipse

Follow the below steps to start each broker:
(a) To start BrokerA:

- Right click *BrokerCore.java* (inside package `ca.utoronto.msrg.padres.broker`) and choose "Run" → "Run Configuration".
- If you are running PADRES for the first time, choose "Java Application" on the side menu and click "New Launch Configuration" button to create a launch configuration for "BrokerCore."
  - Note: If you are running the BrokerCore later on, this configuration will be already available. Modify this configuration as you see fit (also described below).
- In the "Arguments" tab, in the "Program Arguments" box, provide `-uri socket://localhost:3000/BrokerA -n socket://localhost:4000/BrokerB` (Note: to run brokers using the RMI binding replace `socket` with `rmi` in both URIs)
- In the "Arguments" tab, in the "VM Arguments" box, enter the following:
  `-Xms128m -Xmx512m -Djava.security.policy=etc/java.policy`

(b) To start BrokerB, repeat the above steps with the "Program Arguments" as `-uri socket://localhost:4000/BrokerB` (Note: similarly, replace `socket` with `rmi` in the URI to instantiate the broker with RMI binding).

Notes:

- Option `-uri` is used to specify the URI of the broker and option `-n` is to specify the neighbors.
- A broker URI must have the format of `<comm_protocol>://<hostname>:<port>/<broker_id>`. The `comm_protocol` can be either 'rmi' or 'socket'; `hostname` can be the DNS name or the IP address of the machine where the broker is instantiated; `port` is the port number where the broker is listening at for incoming connections; and `broker_id` is the unique identifier of the broker.
- Only one broker can listen at a particular port of a given host.
- Broker ID must be unique and should not contain the '-' (hyphen) character.
- The neighbors are given in a comma separated (with no space) list of broker URIs.
- When two brokers are connected in the overlay, it is enough to specify the neighborhood relation only at one side (in the above example, only BrokerA uses the `-n` option).
- There is no stipulated order in which the brokers have to be started. Brokers will detect whether the specified neighbors are alive by periodically sending connection requests.

5.1.2 From Shell

Use the `startbroker` script located in the `bin/` directory. Follow the instructions in the PADRES *User Guide* for more information.

5.2 Quitting BrokerCore

- If started from eclipse, kill the process using the normal "kill button" (a red square button in the Console pan).
- If started using the `startbroker` script, use the `bin/stopbroker` script to stop brokers as described in the *User Guide*.

5.3 Running GUIClient

This section describes how to run PADRES GUI clients using an example in which two clients are connected to the overlay of brokers described in Section 5.1. ClientA acts as a publisher and is connected to BrokerA and ClientB acts as a subscriber and is connected to BrokerB. There is no difference between starting a publisher and a subscriber; they differ depending on what commands are fed via their interface. Below GUIClient is used in the example, but you can choose to invoke other clients (Section 12).

5.3.1 From Eclipse

To start the clients follow the steps below:
(a) To start ClientA:

- Create a Java Application run configuration as described for running BrokerCore (see Section 5.1), but using the *GUIClient.java* inside package `ca.utoronto.msrg.padres.tools.guiclient`.
- In the "Arguments" tab, in the "Program Arguments" box, provide `-i ClientA -b socket://localhost:3000/BrokerA` (Note: replace `socket` with `rmi` in the URI if brokers are instantiated with RMI binding)

- in the "Arguments" tab, in the "VM Arguments" box, enter the following:
  `-Xms128m -Xmx512m -Djava.security.policy=etc/java.policy`

This should bring up a GUI window interface which accepts publish/subscribe commands from the users. Refer the PADRES user guide for the list of commands available.
(b) To start ClientB, repeat the above steps with the "Program Arguments" as:
`-i ClientB -b socket://localhost:4000/BrokerB`. (Note: replace `socket` with `rmi` in the URI if brokers are instantiated with RMI binding)

5.3.2 From Shell

Use the `startclient` script in the bin directory as described in the *User Guide*.

5.4 Quitting Client

Simply close the GUI window.

5.5 Running PadresMonitor

In its core, the PADRES monitor is also a PADRES client and therefore, the steps for running the monitor is the same as a PADRES GUIClient, except you do not give any program arguments when starting a monitor (you still need to use the same VM arguments). The main class for the PADRES monitor is *MonitorFrame.java* located in package `ca.utoronto.msrg.padres.tools.padresmonitor`. Once the monitor GUI window appears, use menu option "Main → Connect to Federation" to provide the details about the broker you want to connect the monitor to. You can use `bin/startmonitor` script to start the monitor from the shell prompt. Detailed documentation about PADRES monitor can be found in the Section 13 and in the user guide.

6 BrokerCore

The `BrokerCore` class in package `ca.utoronto.msrg.padres.broker.brokercore` is the heart of a PADRES overlay. It instantiates a content-based router with the necessary sub-components. The sub-components include the core components such as content-based matching engine, content-based router, communication interface, input/output message queues, and other auxiliary components to support advanced/optional features such as subscription covering, load balancing, and secure communication.

6.1 Architecture of PADRES Brokers

Figure 1: Architecture of a PADRES broker.

The architecture of the PADRES broker is shown in Figure 1. All the messages received at the communication interface are passed on to the *input queue*. The input queue sequentially processes the messages, sending them through the *router* one by one. For each incoming message, the router can produce multiple outgoing messages. They can be duplicates of the incoming message addressed to different brokers or they can be duplicates of some previously stored messages. For example, an incoming advertisement can trigger some stored subscriptions to be forwarded towards the originator of the advertisement. Each of the messages generated by the router can be placed in different *output queues* allocated for different destinations. These

destinations can be external to the broker such as another broker or a client, or internal such as the *controller* or *heartbeat subscriber*. Output queues each run a separate thread of execution and handle the delivery of the messages to the respective destinations.

The router consists of a *preprocessor*, a *matching engine*, a *forwarder*, and a *post-processor*. The preprocessor selectively modifies the predicates of the advertisements/subscriptions and attribute-value tuples of the publications to assist different functionalities of the broker (for example, routing in cyclic overlays as described in Section 11.4.) The matching engine matches publications, subscriptions and advertisements against each other so that the router can decide to where each message has to be forwarded. It is the job of the forwarder to check the matches produced by the matcher and produce forwarding messages. For each incoming message, the forwarder can produce multiple messages to be disseminated. The forwarder also makes sure that the newly generated messages are properly addressed to the correct destinations; it can also make some load-balancing decision on cyclic networks (see Section 11.4.) The post-processor analyzes the messages generated by the forwarder and applies further modification to improve messaging efficiency (see Section 11.2 and Section 11.3.)

Component	**Operation**	**Package**
CommSystem	The communication layer, responsible for accepting and making connections from/to brokers and clients, and receiving and sending pub/sub message.	`ca.msrg.utoronto.padres.common.comm`
QueueManager	to distribute the incoming and outgoing messages to different internal queues.	`ca.msrg.utoronto.padres.broker.brokercore`
InputQueueHandler	to handle incoming messages. This encloses the operation of router/matcher/forwarder.	`ca.msrg.utoronto.padres.broker.brokercore`
Router	to provide the content-based routing functionality.	`ca.msrg.utoronto.padres.router`
SystemMonitor	to monitor the performances of the broker and generate publications with the observed data.	`ca.msrg.utoronto.padres.broker.monitor`
Controller	to control the operation of the broker. It accepts commands from the publications it receives and takes relevant actions.	`ca.msrg.utoronto.padres.broker.controller`
HeartBeatPublisher	to periodically publish heart beats, based on which the neighbor brokers detects the availability of a particular broker.	`ca.msrg.utoronto.padres.broker.brokercore`
HeartBeatSubscriber	to subscribe and listen to neighbor-brokers' heart beats	`ca.msrg.utoronto.padres.broker.brokercore`
ConsoleInterface	to provide a shell-like CLI to interact with the broker	`ca.msrg.utoronto.padres.broker.management.console`
ManagementServer	similar to the above, but to provide the interface via a web interface	`ca.msrg.utoronto.padres.broker.management.web`

Table 2: PADRES broker components and respective Java classes.

Table 2 lists the primary broker components and their functionalities. In addition to the above core

components, brokers may possess other components to help with specific non-core functionalities. For example, the *system monitor* is not a part of the essential broker components but it is always instantiated with the broker. The instantiations of other auxiliary components are controlled by the configuration options provided via the command line options or the broker configuration file. *Note that* these components must be instantiated and started in a specific order and programmers must pay attention to this matter when modifying the `initialize()` method of the `BrokerCore` class.

7 Communication Interface

Figure 2: PADRES Communication Layer.

Figure 2 illustrates the interactions between the PADRES components (broker/client) and the communication layer. Both the PADRES broker and client use the same code for the communication system. The only difference is that a broker requests the communication layer to create a *listening communication server*, while the clients always make *outgoing connections* to an existing servers (*a running broker*). The unified Java class structure is detailed below. For more details, check the Java doc notes in the source files.

CommSystem This is the communication layer exposed to the PADRES components. A broker creates this object and requests from it to create a listening server that accepts incoming connections and messages. It is possible to ask the CommSystem to invoke more than one listening server. However, the first server created by the CommSystem is also set as the default server. When operations are requested of a server without particularly specifying a server URI, the operations are handled by the default server. CommSystem also provides an API to create MessageSenders (see below).

CommSystem.CommSystemType An enum type provides two constants: `RMI` or `SOCKET`. This has to be extended with more types as more communication protocols are implemented and supported.

CommSystem.HostType Another enum type is used to identify the type of a PADRES entity. Currently two types are supported: `SERVER` or `CLIENT`.

NodeAddress It is an abstract class to define a generalized data structure to keep the details about a server address (e.g. hostname, port number, etc). The `addressFormat` string in this class is used to define the format of the protocol-specific address. In the base class, this is initialized to null, therefore each child class has to define this. Furthermore, `addressFormat` has to be defined in Regular Expression.

CommServer This is the parent abstract class for all the communication servers implementing different communication protocols. As mentioned above, only the brokers invoke `CommServer` in the CommSystem. The primary function of this base class is to implement APIs to register Message and Connection listeners. The actual meat of the communication server is implemented in child classes extended from this.

ConnectionListenerInterface This specifies the protocol-agnostic interface of a connection listener object. The connection listener object should register itself with a server (see `CommServer` above) so that it can be informed when a new connection is made or an existing connection is broken. In PADRES, `OverlayManager` is implementing this interface. Even though the interface definition is general enough, in the current PADRES design, only the making and breaking of connections with clients are informed to the connection listener.

MessageListenerInterface This specifies the interface of a message listener object. A message listener should register itself with a message receiver so that the message receiver can inform the message listener of new messages using the interface. Note that the message receiver can be anything: it can be a communication server or it can be another message listener. In a PADRES broker, the `brokercore.QueueManager` implements this interface. This interface is communication protocol-agnostic.

QueueHander This abstract class provides a threaded implementation of processing messages from a queue. It consists of a `MessageQueue` (see below), on which it performs a blocking remove and process it via the `processMessage()` method. This is an abstract method, which has to be implemented by the classes extending the `QueueHandler` class.

MessageQueue Just a data structure to store an unbounded number of messages.

MessageSender An abstract class to define the interface of an object to connect and send messages to remote entities. Each communication protocol must provide its own implementation of this abstract class. A `MessageSender` can be created either to communicate with a broker or with a client. When a `MessageSender` is created to communicate with a broker, it is constructed with the URI of the remote broker and use the `connect()` API to establish a connection to the broker. In PADRES, the `OverlayManager` explicitly asks the CommSystem to create such `MessageSender`s. The `MessageSender` to communicate with a client is not created actively like this; but created reactively when a server receives a connection request from a client. Once a `MessageSender` object is created, its `send()` method can be used to send a message to the remote entity.

To Brokers		**To Clients**
From Brokers	**From Clients**	**From Brokers**
Created by `OverlayManager`	Created by Client	Created by `CommServer`
Using CommSystem.getMessageSender(...)		Upon receiving a connection from a client
Uses unidirectional connect()	Uses bidirectional connect(...)	does not use connect()
Uses send(...)		

Table 3: Scenarios of using `MessageSender`.

OutputQueue Extended from `QueueHandler`, it also contains an active `MessageSender` to a remote entity (it can be either broker or client). The `processMessage()` API implemented here uses this message sender to send the messages in the message queue to the remote entity.

CommunicationException The exception class thrown by the entities in the communication layer

7.1 RMI-based Communication System

This uses Java Remote Method Invocation (RMI) to implement the communication system.

7.1.1 RMI Server

- RMI servers are registered as RMI entities with an RMI registry. In PADRES, it is the brokers who creates RMI servers (`RMIServer` extended from `CommServer`)
- The URI for an RMI server has the format of: `rmi://<registry_host_name>:<registry_port#>/<rmi_entity_id` where `registry_host_name` is the host name where the RMI registry is located, `registry_port#` is

the port number at which the registry is listening, and the `rmi_entity_id` is the identifier of the entity (server) that is to be registered with the registry.

- The format is defined in `RMIAddress` class (extended from `NodeAddress`).
- There can be multiple RMI registries in the system, as many as one per RMI entity.
- It is possible to register multiple RMI entities (servers) with a single RMI registry.
- If the registry is set to run on a remote machine, it is the responsibility of the administrator to make sure the registry is up and running. RMI server can detect when a registry is not available and report it, but if the registry fails after an entity is registered, the behavior of that entity is undecided after the failure.
- If the host name of the registry is localhost and if the registry is not up, the RMI entity will invoke the registry by itself and register with it. Subsequently, other RMI entities can register themselves with this registry. In this case, special attention has to be given to the first entity, because if it shuts down, the registry shuts down as well and the behavior of the other entities become indeterminate.
- The identifier of an RMI entity must be the same as the Broker ID and must be unique in the system. The administrator has to manually ensure this uniqueness.

7.1.2 RMI Client

- An RMI client can be either a PADRES client or a PADRES broker making a connection to another broker.
- They are represented by `RMIMessageSender`.
- The `RMIMessageSender` created for client→broker connection are special. They use a `RMIMessageListener` which is passed on to the server while making the connection. The `sendTo()` method implemented by the `RMIMessageSender` directly calls the `receiveMessage(...)` API of the `RMIMessageListener` to send a message back to a PADRES client.
- Note that `RMIMessageListener` implements the `RMIMessageListenerInterface`, which is functionally very different from the `MessageListenerInterface` (even though looks the same in name and structure). While the `MessageListenerInterface` is usedfum3(the) Ter whilks is usedfum3(the) Ter whilks

RMIServer Implements the above interface and extends the `CommServer`. This is created by the CommSystem when a RMI-based server is requested by a broker.

RMIMessageSender Extends `MessageSender` and implements RMI-specific routines for the communication between broker→broker, client→broker, and broker→client. A special constructor is provided here that uses `RMIMessageListener` to support broker→client communication.

7.2 Socket-based Communication System

This implementation uses TCP socket-based communication for the PADRES communication layer.

7.2.1 Socket Server

Analogous to the RMI server, socket servers (`SocketServer`) are created one per broker. A socket server is created in the same thread as the broker, but instantiates a connection listener (`SocketConnectionListener`) on a separate thread that listens for incoming connections (from clients or other brokers). For every incoming connection, a client connection (`SocketClientConnection`) object is instantiated in a separate thread which responsible for handling incoming messages. Despite running in multiple threads, all the server related components will shutdown when the server's `shutDown()` method is called.

7.2.2 Socket Client

Similar to the RMI clients, socket clients use `SocketMessageSender` objects for communicating with brokers (to be precise, brokers' socket servers). A socket message sender is instantiated in the same thread as the output queue for a particular broker. Still, each socket message sender creates a `SocketClientConnection` (same class used in socket server above) on a separate thread to handle the notifications from servers as well as reply messages for messages sent. When a reply is required for a message sent to a broker, the message sender locks its execution thread and waits for a notification from the socket client connection thread (see below for *"Asynchronous vs. Synchronous Communication"*).

7.2.3 Socket Implementation Details

SocketAddress Extends `NodeAddress` and keeps track of the information about a socket server (URI, ID, etc).

SocketClientConnection Used in both brokers and Clients, it is the class (always executed in a separate thread) responsible for handling incoming messages.

SocketConnectionListener Always run in a separate thread and accepts incoming socket connections for the server (Broker). Note that this is very different in functionality from the `ConnectionListenerInterface`, despite the similar name.

SocketMessageSender Initiates connections and is responsible for sending all messages for both brokers and clients. However, it acts very different when used with broker and clients (different child methods are called).

SocketPipe It encapsulate the socket read and write and presents an object based i/o to the underlying wire protocol. When we change the wire protocol (see below *"Possible Improvements"*), the work will be done mostly in this class.

SocketServer Extends `CommServer` and provides the access point to a broker. See above *"Socket Server"* for details.

7.2.4 Asynchronous vs. Synchronous Communication

In the case of socket-based communication, synchronous means that, when a server receives a message from a client, it assigns a new messageID to the message and sends it back to the client. Asynchronous, on the other hand, means that when a server receives a messages from a client, it doesn't send anything back.

All of the classes discussed earlier are provide synchronous functionality. Also included are the asynchronous equivalent of these classes, `AsyncSocketClientConnection` and `AsyncSocketMessageSender` for future reference. They extend `SocketClientConnection` and `SocketMessageSender`, respectively. But, they are not used in the system right now, but left it for future use.

7.2.5 Possible Improvements

Currently, the entire `Message` object is sent between PADRES components as Java serialized object. In the future, we will implement our own wire-protocol in order to:

- Enable communication between Java-based brokers and non-Java clients.
- Improve the speed of communication compared to RMI-based scheme. This may involve overriding the `serialize()` method for the `Message` class might achieve this without implementing our own wire-protocol.

7.3 Threads in the Communication Layer

This section describes the threading model in broker and client implementations.

7.3.1 In Brokers

- A `MessageSender` operates in the same thread of the `OutputQueue` that owns it. There is one `OutputQueue` per client or another broker a broker is connected to.
- One thread for the connection server which accepts connections from clients and other brokers. When a connection is made, the following shows the path each protocol takes:
  - Socket protocol: at `SocketConnectionListener.run()` → `SocketClientConnection()` → `SocketClientConnection.notifyConnectionListeners()` → `OverlayManager.connectionMade()`.
  - RMI protocol: `RMIServer.registerMessageListener()` → `OverlayManager.connectionMade()`.
- One thread per each connection made (to client or broker). The following shows the path each message takes in different protocols:
  - In socket protocol: receive at `SocketClientConnection.run()` → `QueueManager.notifyMessage()` → `QueueManager.enQueue()`.
  - In RMI protocol: `RMIServer.receiveMessage()` → `QueueManager.notifyMessage()` → `QueueManager.enQueue()`. Note that even though the method calls listed hear provides an illusion that RMIServer thread is used for incoming message processing as well, when the external entity is registered with the RMIServer, the internal working of RMI produces a separate thread for each registered external entity.

7.3.2 In Clients

- There is one `MessageSender` per the broker the client is connected to. All the `MessageSender` operates in the same thread as the client. i.e. each pub/sub operation is a blocking operation. (Probably this has to change if we go along with the idea of a client interacting with multiple brokers).
- There is a separate thread for each broker the client is connected to for processing incoming messages. The processing of incoming message is done in the same thread as of the `SocketClientConnection` or `RMIMessageListener` object.
  - In socket protocol, the operation sequence is: receive at `SocketClientConnection.run()` → `MessageQueueManager.notifyMessage()` → `Client.processMessage()`.
  - In RMI protocol, it is: `RMIMessageListner.receiveMessage()` → `MessageQueueManager.notifyMessage(` → `Client.processMessage()`.

7.4 Sending Messages: Notes on Return Values and Communication Latency

When a broker sends a message to another broker or client, it does not need to wait for a return message - the lower layer (TCP) will make sure the message is delivered or will throw an exception. However, it is not the case when a client sends a message to a broker. Due to a backward compatibility issue, the IDs of all messages have to be set my the first broker the message touches (the client's host broker). Even though a client produces a message with a client-specific message ID, this ID will be changed at the broker who first receives the message. Now, the client has to know the new ID of the message, so that the client can later use it for unsubscribe/unadvertise the subscriptions/advertisements it issued earlier. Therefore, when a message is sent from client to broker, the broker-modified message ID has to be sent back to the client. This requires a two-way communication to be completed for every successful client→broker communication. Note that, you can still avoid this two-way communication, if it is a publication that is being sent, because there is no unpublish operation in effect in PADRES.

In summary, you need two-way communication for a sending a message, if it is a client → broker communication and the message being sent is either advertisement, subscription, or composite subscription. For all other message communication scenarios, just sending the message is enough.

Even though, this fact is exploited in the two communication protocols (in the implementations of `MessageSender.sentTo()` method), only the socket protocol is benefited from this. Since the RMI protocol always assume a remote method invocation (which implicitly assume a return value), even though the return value of the remote method is void, we don't see a speed up in transmitting publications.

Other than the near 40% speed up in publication dissemination, the communication latency experienced with the socket protocol is not different from the latency seen with RMI protocol. It is most probably because the socket protocol uses the object output/input streams which uses the same Java object serialization used by the RMI scheme. Probably developing a PADRES-specific wire protocol can solve this issue. But this a hypothesis that needs to be studied first in more detail before put into action.

8 PADRES Router

The router is consist of four components: preprocessor, matcher, forwarder, and postprocessor. A message that is sent into the router will go through each of these components in sequence.

The router is implemented as a factory class. Currently, depending on the matching engine specified in the broker configuration (see *User Guide*), the `RouterFactory` class will return either `ReteRouter` or `JessRouter`. Still, for now, both Rete and Jess routers share the same above mentioned subcomponents

except the `Matcher` (which is either Rete-based or Jess-based). In order to support future implementation variations, the subcomponents are implementing functional interfaces.

Note that, even though the code supports `JessMatcher` the required Jess libraries are excluded from the distribution (due to licensing issue) and must be downloaded separately. In absence of Jess libraries, if one tries to instantiate that, the code will crash. If one do possess proper license, they can remove the hard-code "short-circuit" in the `RouterFactory` class to invoke `JessRouter`.

8.1 Matching Engines

This section describes the Jess and Rete matching engines in detail.

8.1.1 Jess Matching Engine

PADRES originally used the Jess library (a rule-based engine implementing Rete algorithm) to produce the matching engine. However, due to licensing issues MSRG decided to implement its own matching engine, which is called *NewRete* (described below). Presently, the code base allows to choose either one of these matching engine to be invoked with PADRES by specifying the desired engine at the `padres.routerfactory` option in the broker configuration file. However, since the Jess library is not included in the code base released to the public, if "Jess" option is chosen for the `padres.routerfactory`, an exception will be thrown at run-time stating *"Jess is not supported"*. In addition to choosing the "Jess" option for the configuration, you have to download the Jess library on your own and place the `jess.jar` file in the `<project_dir>/lib/` directory in order to use the Jess library with the PADRES code base; no other changes are necessary. Note that, currently only version 6.0 of the Jess library is tested with PADRES and most probably you have to buy a license to use this.

8.1.2 NewRete Matching Engine

The objectives of *NewRete* are to replace the Jess-based matching engine in PADRES with a matching engine we control and to extend PADRES to allow the user to select among a set of matching engines. *NewRete* is an implementation of the Rete algorithm adapted to pub/sub routing and matching. This first design is crude mainly aiming to meet the objective of decoupling PADRES from Jess.

Figure 3: Rete Network.

The Rete network is represented as a graph of nodes (Fig. 3 illustrates a sample Rete network). It consists of *1-input nodes* and *2-input nodes*. 1-input nodes includes: the *root node*, *class nodes*, *attribute nodes*, *dummy node* and *terminal nodes*. The root node is the entry point of the Rete Network. Class nodes represent the `class` predicates in PADRES messages. The `class` predicate is a mandatory predicate in PADRES messages. Attribute nodes are predicates in subscriptions and advertisements. Currently, we support Double, Long, String and Time types. It is easy to extend to other types in attribute nodes. A dummy node is used in case an atomic subscription is part of a composite subscription. PADRES's support for variable binding, and variable matching states for atomic subscriptions are maintained in dummy nodes as well. A terminal node represents an atomic or composite subscription in the network. It is the end node of a matching path. If a publication message reaches a terminal node, that means the subscription represented by the terminal node is matched. A *join node* is the only 2-input node we have in *NewRete*. It is used for composite subscriptions. The left and right parents could be other 1-input nodes or 2-input nodes. Fig. 4 shows the details of a join node. The left and right memories in a join node are a list of full match elements. Each full match element is supposed to store information about the publications that constitute a full match

of the sub-expression represented by the left or right parent. It contains the IDs of the publications that constitute the full match of the sub-expression, as well as a list of variable name to attribute value mappings. The join node also contains 3 additional static data: the relevant variables in the left parent memory, this is a mapping from the variable name to attribute name; the relevant variables in the right parent memory, this is a mapping from the variable name to attribute name; and the variables to correlate at this join node. This is a list of variable names. Note that the 3rd static data (`CorrelationVariable`) is computable from the `LeftVariableMap` and `RightVariableMap` but is stored only to make the join processing faster.

Figure 4: Join Node in NewRete.

In each `NodeJoinUnify` class, we have an eventConsumer which specify the consumption policy. To add new consumption policies, you can create a new policy class which extends `DefaultPolicy`. In the new policy class, implement the following method:

```
public boolean consumeEvent(NodeJoinUnify node, String otherMem,
MemoryUnit mem, int matchCount)
```

Then the `nodeJoinUnify` class can use the new policy class to consume events in the left/right memory.

Subscriptions are inserted in subscription arrival order to build the Rete network. When a new atomic subscription comes, we starts from the Root nodes. The child node of the root node is the class node (`NodeTClass`). If the new coming subscription's class is not one of the child node of the Root node, then we create a new `NodeTClass`. Otherwise, we will share the class node. Class nodes have children which are attribute nodes (`NodeTAttr`). Each `NodeTAttr` represents a predicate. As shown in the Rete network, a terminal node represents a subscription (both atomic and composite). Now let's explain the network with examples:

```
S1 = [class,eq,car],[color,eq,red],[year,eq,2002]
S2 = [class,eq,car],[color,eq,red],[year,eq,2003]
S3 = [class,eq,car],[type,eq,BMW],[year,eq,2003]
S4 = [class,eq,stock],[name,eq,IBM]
S5 = [class,eq,insurance],[price,<,150]
CS = {S2 & S3} & S5
```

At first, the Rete network is empty. When the first subscription `S1` comes, we create a `NodeTClass`, two `NodeTAttr`, and a terminal node for `S1`. The two `NodeTAttr` nodes are ordered alphabetically. The second subscription `S2` shares two predicates with `S1`. So no `NodeTClass` nodes is created. The new predicate is inserted as the child node of `NodeTAttr` (`[color,eq,red]`). But not all common predicates can be shared. For example, in `S3`, `[year,eq,2003]` is the same as in `S2`. Since in the predicate list we follow the alphabetic order, we create a new `NodeTAttr` for (`[year,eq,2003]`) following `NodeTAttr` (`[type,eq,BMW]`).

For each atomic subscription, the `NodeTAttr` nodes are ordered alphabetically. If there are other classes, new `NodeTClass` nodes are created as the child node of the Root. To insert a composite subscription, we first identify each atomic part. The atomic parts are inserted as atomic subscriptions, and join nodes are added to connect them.

A publication is passed through the whole Rete network from the Root as far down as possible. If a Terminal node is reached, the subscription associated with the Terminal node matches. The Rete is traversed depth first order. At each node the test is applied, if the test result is false, propagation stops and the depth-first traversal continues. If the test is true, traversal downward the Rete continues. At Join nodes, tests are

applied, if the corresponding value on the other input is available, otherwise, the publication is stored as partial matching state. The depth first traversal continues.

The *publication routing table* (PRT) and the *subscription routing table* (SRT). We use two Rete networks for the two: PRT(S-Rete) and SRT(A-Rete). Subscriptions are inserted in S-Rete, and publications are processed via S-Rete. Advertisements are processed via S-Rete to route existing subscriptions. Advertisements are inserted in A-Rete, and subscriptions are processed via A-Rete. Publications are processed via A-Rete to confirm publications against advertisements.

9 PADRES Messages

PADRES has seven different message types: subscription, composite subscription, advertisement, publication, unsubscription, uncomposite subscription, and unadvertisement (the syntax of these messages are described in the *User Guide*). Each of these is composed of data and message parts: for example, a `Subscription` is contained in a `SubscriptionMessage`. All the messages are subclass of `Message` class.

The messages can be programmatically produced or, given the textual representation of a message (for example, `s [class,eq,stock][company,eq,'IBM']`), the message parser (Section 9.1) can produce the relevant message.

9.1 PADRES Message Parser

We use Java Compiler Compiler (JavaCC) to generate message parser for Padres. JavaCC is the most popular parser generator for use with Java application. JavaCC will read a description of a language and generate code, written in Java, that will read and analyze that language. For the common issues about JavaCC that you are interested in, please refer to JavaCC-FAQ[3].

The message parser is located in `ca.utoronto.msrg.padres.common.message.parser` package. The grammar specification of PADRES messages is defined in `MessageParser.jj`. If `MessageParser.jj` is error free, it can be complied through JavaCC to produce a number of Java source files. Modifying any generated files must be generally avoided.

If you update the grammar file and generate new Java source files, you must commit only the modified grammar files into CVS (without the generated Java file). The ant builder will run JavaCC and re-generate the Java files for the end user.

Running PADRES with JavaCC requires certain JavaCC options to be set correctly in the project properties. Please refer Section 4 for more details. If you are using Eclipse for PADRES development, please make sure that you have the JavaCC plug-in installed for Eclipse.

9.2 Time-to-Live in Messages

Time-to-live (TTL) in PADRES messages is used to control the dissemination distance of a particular message. To be precise, TTL is the hop count the particular message is supposed to travel. Each **Router** will decrement the TTL value of a message and when the its value is zero the message is simply dropped by a router. TTL is useful in control messages; for a example, TTL value of the `CONNECT_REQ` messages (Section 11.1) is set to 1. By default, TTL values of the messages are set to a negative number which denotes an unbounded propagation.

[3] http://www.engr.mun.ca/ theo/JavaCC-FAQ/

9.3 Parser Grammar

The main parser source file is `MessageParser.jj`, a JavaCC grammar file which will be compiled to actual `.java` code during the build process. The current ant builder (located in `$PADRES_DIR/build.xml`) automatically compiles and produces the java files. The Padres parser also internally uses `TokenReturner.jj`. Note: Only the main and auxiliary `.jj` files (as well as the factory class - see below) are distributed. The formal Padres message grammar is as follows:

```
CLASSATTRVALPAIR= "[" , CLASS , COMMA , [ AVALSTR | ANAME ] , "]"
ATTRVALPAIR = "[" , ANAME , COMMA , [ IVARIABLE | SVALUE | ANAME | AVALDOUBLE | AVALLONG ] , "]"
SVALUE = [ AVALSTR | SVARIABLE ]
IVALUE = [ AVALDOUBLE | AVALLONG | IVARIABLE ]
ATTRIBUTEOPERATORVALUETRIPLE = "[" , ANAME , COMMA ,
 [
  ( [ SOPERATOR | EQ ] , COMMA , [ SVALUE | ANAME ] )
  |
  ( IOPERATOR , COMMA , [ IVARIABLE | AVALLONG | AVALDOUBLE ] )
  |
  ( ISPRESENT , COMMA , [ SVALUE | ANAME | IVARIABLE | AVALLONG | AVALDOUBLE ] )
 ] , "]"
CLASSATTRIBUTEOPERATORVALUETRIPLE = "[" , CLASS , COMMA , EQ , COMMA , [ AVALSTR | ANAME ] , "]"
ADVERTISEMENT = SUBORADVERTISEMENT
SUBSCRIPTION = SUBORADVERTISEMENT
PUBLICATION = CLASSATTRVALPAIR , ( COMMA , ATTRVALPAIR )* , SEMICOLON , ( TIMEFORMAT , SEMICOLON
SUBORADVERTISEMENT = CLASSATTRIBUTEOPERATORVALUETRIPLE , ( COMMA , ATTRIBUTEOPERATORVALUETRIPLE )
COMPOSITESUBSCRIPTION = TERM , ( COPERATOR , TERM )*
TERM = "{" , SUBORADVERTISEMENT , "}" | "{" , COMPOSITESUBSCRIPTION , "}"
SOPERATOR =  "str-le" | "str-lt" | "str-gt" | "str-ge" | "str-prefix" | "str-postfix" | "str-cont
IOPERATOR = "=" | "<" | ">" | "<=" | ">=" | "<>"
COPERATOR = "\&" | "||"
ANAME = [ ALPH | "_" | "-" | ":" | "/" ]
 |
 ANAME , DIGIT
TIMEFORMAT = ANAME , ( " " , [ ANAME | AVALZEROINT | AVALLONG | ([ ":" | DIGIT ])* ] )*
AVALZEROINT = "0" , ( DIGIT )+
AVALSTR = "\'\\'"
 |
 "\\"\\""
 |
 "\\'" , [ ALPH | "|" | "\&" | "_" | "-" | "[" | "]" | " " | "{" | "}" | "<" | ">" | "=" | DIGIT
 ([ ALPH | "\$" | "|" | "\&" | "_" | "-" | "[" | "]" | " " | "{" | "}" | "<" | ">" | "=" | DIGIT
 |
 "\\"" , [ ALPH | "|" | "\&" | "_" | "-" | "[" | "]" | " " | "{" | "}" | "<" | ">" | "=" | DIGIT
 ([ ALPH , "\$" | "|" | "\&" | "_" | "-" | "[" | "]" | " " | "{" | "}" | "<" | ">" | "=" | DIGIT
AVALLONG = NONZERODIGIT , ( DIGIT )*
 |
 "0"
AVALDOUBLE = NONZERODIGIT , ( DIGIT )* )*AVALDOUBLE  , ( DIGIT )*
```

```
 "0." , ( DIGIT )+
 |
 "." , ( DIGIT )+
SVARIABLE = SVAR
 |
 "\\'" , SVAR , "\\'"
 |
 "\\"" , SVAR , "\\""
IVARIABLE = [ IVAR | FVAR ]
 |
 "\\'" , [ FVAR | IVAR ] , "\\'"
 |
 "\\"" , [ FVAR | IVAR ] , "\\""
IVAR =   "\$I\$" , [ ALPH | "_" | "-"] , ( [ ALPH | "_" | "-" | DIGIT ] )*
SVAR =   "\$S\$" , [ ALPH | "_" | "-"] , ( [ ALPH | "_" | "-" | DIGIT ] )*
FVAR =   "\$F\$" , [ ALPH | "_" | "-"] , ( [ ALPH | "_" | "-" | DIGIT ] )*
ALPH = [ "a"-"z" | "A"-"Z" ]
DIGIT = [ "0"-"9" ]
NONZERODIGIT = [ "1"-"9" ]
CLASS = "class"
COMMA = ","
EQ = "eq"
SEMICOLON = ";"
COLON = ":"
ISPRESENT = "isPresent"
```

9.4 Parser Integration

Padres message parser, once compiled, is usable within the rest of the project through the MessageFactory.java class which instantiates a static and private internal message parser. The factory class takes care of parser initialization and provides a ready-to-use abstraction for the parser. This is provided through the following API shown Table 4.

9.5 Valid Message String Representations

Various message objects can be converted to/from string representation via appropriate calls to `Message.toString()` or `MessageFactory.createXXXFromString(String)`. Table 5 shows sample string representation for different message type. Note that in the Padres message format, the class attribute must always be present.

10 Other Broker Components

In addition to the core components described in the previous sections, brokers include other components to support non-core function; some are critical and others are auxiliary and for the users' convenience.

10.1 Broker Controller

Broker controller is the component that manages pluggable components. Any objects that implements the `brokercore.controller.Manager` interface can be added to the list of pluggable components (*a.k.a.*

Method	Descrition
`Publication createEmptyPublication()`	Returns an empty publication object.
`Publication createPublicationFromString(String)`	Converts the string representation of a publication into an actual pub object.
`Subscription createEmptySubscription()`	Returns an empty subscription object.
`Subscription createSubscriptionFromString(String)`	Converts the string representation of a subscription into an actual sub object.
`Advertisement createEmptyAdvertisement()`	Returns an empty advertisement object.
`Advertisement createAdvertisementFromString(String)`	Converts the string representation of a advertisementinto an actual adv object.
`CompositeSubscription createEmptyCompositeSubscription()`	Returns an empty composite subscription object.
`CompositeSubscription createCompositeSubscriptionFromString(String)`	Converts the string representation of a composite subscription into an actual CS object.

Table 4: PADRES parser API.

managers) that the controller handles. The controller is primarily a secondary message dispatcher between the `QueueManager` and the list of managers. As of PADRES 2.0, there are three managers in use:

- `OverlayManager`: responsible for handling broker/client connections.
- `LifeCycleManager`: responsible for instantiating components required for the broker's optional features. As of PADRES 2.0, it handles only the DB component needed to support historic queries (Section 11.5).
- `ServerInjectionManager`: responsible for injecting arbitrary messages into the broker overlay without a client — mostly used for monitoring purposes.

10.2 Console Interface

Console interface provides a command line interface (CLI) via which users can interact with a broker to query its states and to control its operations. Please refer to the PADRES *User Guide* for the available commands.

Publication:	`p [class,stock],[name,IBM],[price,45]`
Subscription:	`s [class,eq,stock],[name,eq,IBM],[price,>,60]`
Advertisement:	`a [class,eq,stock],[name,eq,IBM],[price,>,50]`
CompositeSubscription:	`cs [class,eq,stock],[name,eq,IBM],[price,>,50]& [class,eq,stock],[name,eq,HP],[price,>,40]`
	`cs [class,eq,stock],[name,eq,IBM],[price,>,50]|| [class,eq,stock],[name,eq,HP],[price,>,40]`

Table 5: Sample string representations of PADRES messages.

The console interface is implemented inside the package `ca.utoronto.msrg.padres.broker.management.console`. When the CLI is enabled, broker instantiates the `ConsoleInterface` class which provides the command prompt, accepts commands from the prompt, and display the results. The commands are processed by the `CommandHandler` class. The primary interface to the command handler is `handleCommand(String cmd)` method which accepts the commands and arguments as a string.

The `CommandHandler` class is created in a way that it is easy to add new commands. Follow the below steps to add a new commands. For example, let's consider we want to add a new command `hello` that accepts a string argument.

1. Implement a new method called `helloWorld(String[] argv)` to handle the command. Note that the arguments to the command is passed through `argv` as an `String` array (it does not include the command, so that the first argument is found at `argv[0]`).

2. Inside the `initializeCommandMap()` method of `CommandHandler` class, add two lines to add new entries into the command–method and command–description maps.

```
commandMethodMap.put("hello", "helloWorld");
commandDescriptionMap.put("hello", "Say hello to the user. Usage: hello [name]");
```

10.3 Web Management Interface

The functionality of the web management interface is the same as the console interface (Section 10.2), but the interface is accessed through a web browser. Please refer the PADRES *User Guide* for more information.

Figure 5: Web management interface of the PADRES broker.

The web management interface is implemented inside the package `ca.utoronto.msrg.padres.broker.management.web`. The architecture of the web management interface is shown in Figure 5. When the interface is enabled, it instantiates a web server. PADRES uses the version 3.1 of the Simple[4] API to create this web server.

This web server in turn instantiates three services, each servicing different incoming HTTP requests: the requests with *http://<hostname>:<port>/broker/* as the prefix are handled by `BrokerService`; the requests for html files and the base URL (*http://<hostname>:<port>/*) are serviced by `PageService`; and all the rest are handled by `FileService`. New services can be added in the `registerServices()` method of `SimpleServer` class. All the services should be a subclass of `simple.http.load.Service`.

[4] http://www.simpleframework.org/

The broker service redirects the incoming commands to the `CommandHandler` and sends the results in an XML packet. The command handler is the same one used by the console interface (Section 10.2).

The interface seen by the users inside a web browser is produced by the `shell.html` located in the `etc/web/management/` directory. The page is rendered and supported by two Javascript files `shell.js` and `ajax.js`. The `shell.js` is responsible for updating the user interface, while the `ajax.js` handles the communication between the user-side interface and broker-side web server. The commands are sent and results are received through XML-based Ajax communication objects.

11 PADRES Operations

11.1 Connection Establishment between Brokers

A bi-direction connection between two PADRES brokers is established as two uni-directional connections between them in different directions. For each uni-directional connection, one broker acts as the server while the other acts as the client. A uni-directional connection is established through the following steps:

1. If a list of neighbors is given to the broker (via CLI or config file), the `BrokerCore` will instruct the `OverlayManager` to create connections to these neighbor brokers. Here, the neighbor brokers will act as connection servers, while the broker initiating the connection will be the connection client.

2. The `OverlayManager` will request a `MessageSender` from its `CommSystem` for each neighbor broker, then call the `connect()` API of the `MessageSender` to connect to the connection server of the neighbor. (The exact implementation of this varies depending on whether the brokers are using socket or RMI protocol). This will create a one-way communication channel between the client and server brokers.

3. When the above is successful, an `OutputQueue` with the created `MessageSender` will be created and placed in the broker's `OverlayRoutingTable` so that the connection between the broker is kept alive (instead of creating a connection for each message-send).

4. Also the `OutputQueue` is registered with the `QueueManager` so that it will know which queue to use if a message is to be send to the particular neighbor broker.

5. Then the `OverlayManager` will send a `CONNECT_REQ` message to the newly connected neighbor as a publication (also an advertisement is produced just before this to support this publication).

6. When a broker receives the `CONNECT_REQ` message, it replies with a `CONNECT_ACK` message. Also, if it does not know about the broker where the message came from, it executes steps 2–5 to create the other uni-directional connection in the two-broker connection.

7. When the `CONNECT_REQ`-originating broker receives the `CONNECT_ACK` message, the connection is considered to be established. The broker will now forward all the advertisements it holds to the newly connected broker.

Note the followings in the above procedure:

- In the connection establishment between two brokers, specifying the connection at one end is enough. Also, it does not matter which broker comes alive first; provided that the other broker becomes active within a reasonable amount of time (determined by connection-retries parameter in the configuration), the connection between the two will be established.

- `MessageSender` is a communication layer implementation and is always uni-directional. Even though the same `MessageSender` class is used for client–broker and broker–broker communication, the exact API used for these two cases a different; especially the `connect()` API.
- As of PADRES 2.0, the `Controller` object acts as the secondary message multiplexer in between the `QueueManager` and `OverlayManager`. Therefore, all the messages destined to `OverlayManager` must specify `BROKER_CONTROL` as the `class` and include the actual message as a `command` predicate. The "commands" to be received at the `OverlayManager` must be prefixed with `OVERLAY-`.

11.2 Subscription Covering

The default setting for subscription covering is `OFF`, both inside the code and in `padres.properties`. When you want to test with covering on, simply change the `padres.covering.subscription` parameter to `LAZY` or `ACTIVE` in the broker configuration file.

Subscription covering is done by the SIENA method of using a poset to maintain a subscription-covering graph with the most general subscriptions at the top of the graph. The structure is named as *scout* in the code base, which originally stood for *Subscription Covering Ordered Utilizing a Tree*.

If subscription covering can be configured to operate in a `LAZY` or `ACTIVE` manner. For instance, if a subscription `[class,eq,'stock'],[price,=,10]` is forwarded, and in the future a more general subscription gets forwarded to the same destination, say `[class,eq,'stock'],[price,>,0]`, the original `price=10` subscription is not unsubscribed in the `LAZY` mode unless the client unsubscribes from the `price=10` subscription explicitly. On the other hand, in the `ACTIVE` mode, the less general (first) subscription is unsubscribed automatically. In other words, difference between `LAZY` and `ACTIVE` modes is that the former does not actively unsubscribe previous subscriptions that are now covered, while the latter this unsubscription place automatically.

Right now, there is no support for composite subscription covering.

11.3 Advertisement Covering

Similar to subscription covering, advertisement covering is also done by the SIENA method of using a poset to maintain a advertisement-covering graph with the most general advertisements at the top of the graph. The Java structure is named as `scoutAD` in the code base, which originally stood for *Subscription Covering Ordered Utilizing a Tree for ADvertisements*.

Advertisement covering operates in a `LAZY` manner — no unadvertisement is issued. For instance, if an advertisement `[class,eq,'stock'],[price,>,10]` is forwarded, and in the future a less general subscription gets forwarded to the same destination (`[class,eq,'stock'],[price,>,15]`), the second advertisement will not be sent out from the broker due to advertisement covering. Furthermore, if an advertisement `[class,eq,'stock'],[price,>,10]` is forwarded, and in the future a more general subscription gets forwarded to the same destination (`[class,eq,'stock'],[price,>,5]`), the original `price>10` advertisement does not get unadvertised unless the client issues an unadvertisement explicitly.

11.4 Routing in General Networks

TODO: how it is done – conceptual description
TODO: how it is implemented – in terms of the code fragments throughout the PADRES code base and what they are about.

11.5 Historic Data Query

PADRES has a capability that allows clients to subscribe to publications from the past. These special subscriptions are called *historic data queries*. Once PADRES receives historic data queries, PADRES converts them into SQL queries. With these SQL queries, PADRES retrieves publications stored in a database whose properties are configured in the `db.properties` file. This file is placed under `<PADRES_HOME>/etc/db` by default. PADRES does not enforce the subscribers to be aware of the database schema for holding publication records and issue valid SQL queries. PADRES automatically converts the historic data queries that are specified in regular PADRES subscription language into SQL queries. In the following, we describe the components that handles database setup and query conversions and processing.

Classes that handle historic data queries are located under `ca.utoronto.msrg.broker.controller.db`. The `LifeCycleManager` of a broker *binds* a database by constructing a `DBHandler` object. `DBHandler` constructs a `DBConnector` object that is responsible for setting up the databases according to the properties specified in `db.properties` and processing SQL queries. `DBHandler` also constructs `SQLConverter` object. The `SQLConverter` object maps a historic data query specified in the PADRES language into a SQL query through the `getAllPubs` function. This `SQLConverter` object also converts publications into SQL and stores them into the database through the `executeQuery` function of `DBConnector`. The tables that hold the publications are initialized through `initDatabase` function of `DBConnector` with the table creation statements specified in `createTableStatements` which reflect the database schema.

11.6 Broker Info Messages

Broker info messages are published by the `SystemMonitor` (via `BrokerInfoPublisher`). These messages contain information about the broker such as its broker ID, URI, neighbors, clients, memory levels, etc. The full set of information included in a broker info message can be found in the `SystemMonitor.getBrokerInfo()` method.

By default, broker info messages are not published periodically. However, one can turn on periodic publishing by using the correct command line argument or property in the broker's configuration file. Another alternative is to leave the broker info at its default setting and force the broker to publish a broker info message by sending it a `[class,eq,'NETWORK_DISCOVERY']` message. Broker info messages can be subscribed to by any client. Current applications that use broker info messages include the GUI monitoring client and PANDA. Both applications will still function even if broker info messages are set not to be published periodically.

12 PADRES Clients

As of PADRES 2.0, all the PADRES clients are extended from the base client class `ca.utoronto.msrg.padres.client.Client`. This superclass itself support full-fledged client operations such as:

- `connect()` and `disconnect()` API calls to connect to and disconnect from given brokers (given in the form of broker URIs).
- publish/subscribe operations (advertise, subscribe, publish, unadvertise, etc).
- `handleCommand()` API call to process textual PADRES commands like `a [class,eq,stock]`.

As described in the *User Guide*, PADRES now provides four different types of clients: `CLIClient`, `GUIClient`, `WebClient`, and `PadresSwingRMIDeployer`. While all of them fundamentally provide the same pub/sub functionality, their user interface differs. All these clients can be invoked using the `startclient` script. Refer the *User Guide* for more details.

Still, if you require a specific functionality in a PADRES client that is not yet implemented, you can implement it on your own by extending the base PADRES client. The next section describes how.

12.1 Extending PADRES Client

The `ca.utoronto.msrg.padres.client.Client` class provides the functionalities for any client that wants to join a PADRES overlay. The `Client` class provides APIs for all the basic pub/sub operations like advertise, subscribe, publish, unadvertise, etc. It also provides APIs to connect and disconnect from PADRES brokers and process textual PADRES messages. One might want to extend this base client in order to provide programmatic access to these APIs, for example, for auto-generating and publishing messages periodically.

There are already few extended classes in the `ca.utoronto.msrg.padres.tools` package if you want to see examples. If the new extended client is to handle additional configuration options, then the `ca.utoronto.msrg.padres.client.ClientConfig` class also has to be extended to accept/process the new options (an example is `ca.utoronto.msrg.padres.tools.webclient.WebClientConfig`). The constructor of the extended client will now accept this extended configuration object instead of the default object. The default `ClientConfig` object is created by reading `etc/client.properties` configuration file. You can use a different configuration file with the same format to create the default or extended configuration object.

The `Client` class provides `connect()` method for connecting to PADRES brokers. When a client use the `connect()` call to connect to a broker a message sender (`ca.utoronto.msrg.padres.common.comm.MessageSender`) object is created for the broker. The internal details of the `MessageSender` class is unnecessary here, but you can use the `send(Message)` call of the `MessageSender` class for pub/sub operations (as used in the Client's pub/sub APIs). The `connect()` method also register a message listener (`ca.utoronto.msrg.padres.common.comm.MessageListenerInterface`) with the created message sender object. When a message is received from the broker, this message listener will be notified via the `notifyMessage()` method implemented by the message listener object.

The message listener interface for the client is implemented by the `ca.utoronto.msrg.padres.client.MessageQueueManager` class. Every client has one instance of the `MessageQueueManager`. For every incoming message, this manager will call the client's `processMessage()` method. In addition, it also acts as the central dispatcher for the incoming messages. Additional message queues (`ca.utoronto.msrg.padres.common.comm.MessageQueue`) can be registered with this central message dispatcher and a copy of an incoming message will be distributed to each such registered queue. Generally, overriding the `processMessage()` method in the extended client class is required to implement new logic for processing incoming messages. However, if the processing of incoming messages is expected to take a long time, it is better to do that on a separate thread instead of blocking the communication thread. For this scenario, it is better to create a processing thread with a message queue and register the queue with the `MessageQueueManager`.

The `Client` class employs the `ca.utoronto.msrg.padres.client.CommandHandler` class to handle the textual commands fed to the client. This is an abstract class and all the actual classes implementing logic for handling textual commands must extend this abstract class. The `Client` class instantiates a `client.BaseCommandHandler` object to handle all the basic textual commands (including all the PADRES message commands). In case you find that the commands supported by this `BaseCommandHandler` class is inadequate, you have to implement your own command handler (extending `CommandHanlder` abstract class) and register the new command handler with the client using the `Client.addCommandHandler(CommandHandler)` method. All the commands provided by all the command handlers are accessed via a single API call: `Client.handleCommand(String)` method. Note that, most likely, a client that programmatically access the pub/sub APIs will not employ textual commands and hence will have no use of any `CommandHandler`.

The `Client` class supports connecting to multiple brokers at the same time. All the pub/sub API calls

can be made specific to a connected broker or all the connected brokers. If no broker URI is specified in a pub/sub API call (which is the usual case in the current operations of PADRES) the default broker will be used. The default broker is the first broker a client is connected to. However, correctness of PADRES operations when a client connecting to multiple brokers is not tested as of PADRES 2.0.

In addition to all the pub/sub APIs, the followings are some important methods in the `Client` class to take note:

`initLog(String)`: protected method to initialize log4j logger. Accept the name of the log directory as input.

`connect(String)`: use this to connect to a broker; accepts a broker URI as input.

`disconnect(String)`: reverse of the above.

`getDefaultBrokerAddress()`: returns the URI of the default broker.

`getBrokerState(String)`: this will accept a broker URI string and return the `BrokerState` of the specific broker. A `BrokerState` object contains the `MessageSender` for the broker (which can be used to send messages to the broker) and the history of all the advertisement, subscription, and composite subscription sent to the specific broker. However, in order to avoid heavy memory consumption, storing the advertisement/subscription history is turned `OFF` by default. Turn it on by setting the `client.store_detail_state` property `ON` in the `client.properties` file.

`addMessageQueue(MessageQueue)`: use this to add an message queue into which a copy of a newly received publication message will be queued. See above for detail.

`processMessage(Message)`: this method is called whenever the client receives a message from the connected broker. By default, this method just stores the last-received message so that it can be retrieved by the `getCurrentPub()` method. Override this method in the extended client if you want to implement additional logic (see `GUIClient` for example).

`addCommandHandler(CommandHandler)`: to add additional command handlers to process new textual commands (example can be found in `CLIClient`).

`handleCommand(String)`: accepts a textual command and process it using one of the `ComandHandler` available. See above for detail.

`shutdown()`: unsubscribe and unadvertise all the subscriptions and advertisements respectively and disconnect from all the brokers. It does not exit the client process; if you want that functionality, override this method in the extended class (example can be found in `CLIClient`).

13 PADRES Monitor

As part of the PADRES project, a specialized client has been developed that can be used to graphically display the overlay network topology (i.e., brokers, clients, and their inter connections) and to interact with the brokers (e.g., inject/trace a message, stop/shutdown/resume brokers, detect failures, etc). The monitoring tool connects to a broker within the deployed PADRES network like a regular client (i.e., with RMI binding), and subscribes to *BROKER_INFO* messages published by individual brokers. It uses the information received in these publications to construct the network topology and display it graphically. Please refer to the PADRES User Guide for information about monitor features and usage instructions.

13.1 Running the PADRES Monitor

13.1.1 From Eclipse

Launching the monitor from the Eclipse IDE is similar to launching a client. The PADRES monitor is implemented in `MonitorFrame.java` (inside package `ca.utoronto.msrg.padres.tools.padresmonitor`). Make sure that, in the *Argument* tab of the run configuration, the following option is added to the *VM arguments* field:

```
-Djava.security.policy=etc/java.policy
```

Note: you don't have to provide anything for *Program Arguments*.

13.1.2 From Shell

The monitor can be launched from the shell using the `startmonitor` script:

```
$ startmonitor
```

14 PADRES Logging

Logging of PADRES messages/exceptions are handled through the log4j external library. This library is included in the `lib/` directory. Only one logger can be created per process, and it is important to note this fact when creating/running test cases. For example, the JUnit test cases instantiate multiple brokers and clients within a single process and therefore they have to share the same logger. Therefore, it is important to create log messages that identify themselves about in which PADRES component they are originated (log4j can identify which Java class creates a log message, but it cannot identify which instantiation of the class generates the message).

PADRES logging structure is mainly configured using the `etc/log4j.properties` configuration file. Details of the configuration file is discussed in the BrokerCore configuration section of the PADRES User Guide. By default the log files are placed in the `/.padres/logs/` directory.

Log4j has a problem of generating log files for all the classes mentioned the configuration file with a file appender option, irrespective of some of those classes have a lower log level than the root logger. It is prevented by the following approach:

- the configuration file never adds a log4j file appender to any class.
- the `ca.utoronto.msrg.common.util.LogSetup.java` class reads the `log4j.rootLogger` log level and other class names listed in the configuration file, and creates a log4j file appender for each class that has a log level equal or higher than that of the `rootLogger`.
- in this way, not only the messages with low log levels are stopped, but also the relevant log files are prevented to be generated.

The `addFileAppender()` and `getClassLogFileName()` methods in the `LogSetup` class provide a way of manually intervening the above process of creating log files. These functions will be needed mostly for JUnit tests. Example of how to exploit these functions can be found in the `TestHeartBeat` test cases of `test.junit.suite*`.

15 Testing PADRES

15.1 Introduction to New PADRES Test Framework

The PADRES test framework allows testing of various feature of brokers and clients. The implementation of the new PADRES test framework that is used for these testcases are located in package `ca.utoronto.msrg.padres.test.junit.tester` and current testcases are distributed under the `ca.utoronto.msrg.padres.test.junit` package. The new test framework is organized around a key interface called `IBrokerTester` and its implementing class `GenericBrokerTester`. In the beginning of each testcase, a number of expected and unexpected broker/client events are registered with the `IBrokerTester` object. At the end of the testcase, the framework automatically examine occurrence of these events and the testcase succeeds or fails accordingly.

15.1.1 Expected and Unexpected Broker/Client Events

A key concept in the new PADRES test framework is that testcases define and register broker/client events that they would like to see take place or would like to ensure that do not occur. These are referred to as *expected* and *unexpected events*, respectively. Examples of these events range from: sending or receipt of messages of given types, destinations, or with containing different predicates, broker's state change (shutdown, start, resume), client's publication delivery events, etc. An unexpected broker/client event is one that must not take place within the execution of a testcase. In other words, the test must fail if such events take place during the exection. For example, a covered subscription must not arrive at a certain broker (i.e., arrival of a covered subscription may indicate incorrect routing and state maintenance and the test must subsequently fail indicating an erroneous execution).

15.1.2 Writing New Testcases

The following is the steps involved in a typical testcase:

- Each testcase instantiates a `GenericBrokerTester` object (which implements `IBrokerTester`) and uses it to register a number expected and unexpected broker events. Event registration can take place in the beginning of the testcase. Alternatively, and indeed more commonly, event registration may be done as the testcase progresses through several stages, each involving the same brokers/clients but with their own set of expected/unexpected events. For example, stage one may involve dissemination of advertisements, stage two involve dissemination of subscriptions and finally stage three tests whether published messages arrive at interested subscribers. At the beginning of each stage, clients and brokers perform operations for that stage, e.g., issue advertisements, subscriptions, or publications. At the end of each stage, the testcase examines whether expected/unexpected events took place or not. If an expected event is missed or an unexpected event has occurred the testcase must fail and exit. Upon success of one stage, on the other hand, the test progresses to the next stage or ends successfully (after the final stage).

- The testcase instantiates specialized subclasses of `BrokerCore` and `Client` (`TesterBrokerCore` and `TesterClient`, respectively) which are created and hold a reference to the testcase's `GenericBrokerTester` (this is passed in in their constructors).

- The testcase then progresses through possibly several stages. In each stage, a number of expected/unexpected events are registered with the `GenericBrokerTester` object. The test framework keeps track of these events in order to ensure that they do or do not take place for the brokers and clients involved.

- At the end of each stage, occurrence of broker/client events are checked via a blocking call to `IBrokerTester.waitUntilExpectedEventsHappen()`. The function automatically returns within either a (default, or specified) timeout. The testcase must use an assertion to check the return value of the function. The assertion succeeds or fails if there is an event that is missed (within a certain timeout), or an unexpected event that takes place (indicating an error condition) the test must fail.

15.1.3 Test Case Organization

There are a number of testcases (each containing many tests) under the `ca.utoronto.msrg.padres.test.junit` package. These testcase test the pub/sub functionality at various levels:

- `TestBroker`: tests the correct initialization of a broker.
- `TestBrokerException`: tests whether brokers throws relevant exceptions under different error conditions.
- `TestClients`: tests pub/sub operations with atomic subscriptions.
- `TestClientExceptions`: tests whether clients throw relevant exceptions under various error conditions.
- `TestComplexCS`: tests pub/sub operations with complex subscriptions.
- `TestHeartBeat`: tests whether heartbeat-based overlay operations are functioning correctly.
- `TestMonitor`: tests whether overlay control via PadresMonitor is operating correctly.
- `TestTwoBrokers`: tests whether pub/sub operations are performed correctly in a simple two-broker overlay.
- `TestMultipleBrokers`: tests whether pub/sub operations are performed correctly in a multi-broker overlay.

There are more testcases in the sub-packages:

- `components`: Contains testcases to check PADRES components (to test property file loading, command line argument processing, and historic query processing).
- `components.scout`: Contains testcases to validate PADRES's subscription covering routines.
- `components.scoutad`: contains testcases to validate PADRES's advertisement covering routines.
- `cyclic`: Contains testcases to test pub/sub functionalities in a cyclic overlay.

All the testcases are launched via the `AllTests` class which accepts two inputs (as JVM arguments): communication protocol (`-Dtest.comm_protocol`) and test version (`-Dtest.version`). The communication protocol defines the protocol to be used in the communication system layer, while the test version decides the system configurations. The following table lists the test versions and related configurations: By default, the test version is 1 and the communication protocol is "socket".

15.2 Running Tests

The tests can be run via the Eclipse IDE, command line, or using the *ant*.

15.2.1 Eclipse

If you are running the tests via Eclipse or command line, please specify the appropriate test version and communication protocol using `-Dtest.version=<version> -Dtest.comm_protocol=<protocol>` JVM arguments (that is, if you want to change the default values). You have to launch each test version manually yourself.

15.2.2 Ant

Using ant can be more convenient. Launch a single test using ant as:

```
$ ant -Dtest=<version> -Dcomm=<protocol> test
```

again, the JVM arguments are optional. You can also run automate running all the test version one after another by issuing:

```
$ ant -Dcomm=<protocol> test-all
```

When using ant, a summary of the test results will be printed on screen and the test-by-test information will be logged into `AllTests-<test_version>.junit.txt` files.

NOTE: Tests in `TestBrokerExceptions` deliberately try to make connection to non-active broker URIs. The test passes only if the connection fails. Make sure the the URIs are actually unavailable; i.e. the URI is valid, but the port (of the URI) is closed. You can check it by:

```
$ nc -zv <ip_addr> <port_no>
```

15.3 Old Test Framework Design Notes

Note that all future test cases must be written in the new test framework (described in 15.1). However, the description in this section is useful since some of existing PADRES testcases are written using the old PADRES test framework which uses string pattern matching against the logged entries to check for occurrence of specific events relevant to each tests case. This is in contrast to the new test framework in which events are defined through the standardized tester interface, *i.e.*, `IBrokerTester` interface. The downside of the old framework which motivated the design of the new test framework was that all testcases needed to be rewritten after even small changes to the string representation of messages. Furthermore, we had observed cases where the pattern matcher in the old framework was unable to distinguish between similar events taking place at different brokers (this is due to the fact that all brokers running in the same JVM would append their logs to the same file).

The old framework testcases are also run through the `AllTests` class by setting up the corresponding broker/client configurations. Note that the client configurations turn on the option to store detailed states, so that all the subscription/advertisement sent by a client and all the publications received by a client is stored for verifications needed by some tests. The `AllTests` class also provides methods for setting up pre-defined overlay patterns. These methods are used by testcases that test the system with multiple brokers. Most of the tests follow a similar pattern:

- Perform an operation.
- Wait for a message to be routed to or out of a certain broker/client or an exception to be thrown.
- Checks the broker/client state or thrown exception to verify the state or the thrown exception is correct.

In order to wait for a message to be routed or the exception to be thrown, the custom-made log4j appender called `MessageWatchAppender` is used. With the help of the `PatternFilter` class, it can specifically look for a pattern in the message path or exception capture. For details, look into these classes. CAUTION: Because of the testcases' dependency on log4j, be careful when you want to change the logging location (in the code) and format (of the message) of the `MessagePath` and `Exception` loggers.

15.4 Testing PANDA

There is no JUnit tests for testing PANDA. You have to run PANDA for a test topology and verify that it works.

15.5 Testing PadresMonitor

There is one JUnit test (TestMonitor) for testing testing the monitor and other testcases sometimes use monitor as an accessory. Still, it is better to test the monitor manually invoking all of its menu operations in a test topology. Please refer the detailed guide to Testing PADRES Monitor for details.

The correctness of the PADRES components is tested using various JUnit test cases, inside package `ca.utoronto.msrg.padres.test.junit`. Even though the unit tests evaluate the correctness of each PADRES features/operations, they mostly create a test PADRES overlay and treat it as a black-box; only a few test cases are created for testing broker/client components such such subscription covering or message parser. For the black-box testing, the assertions are made at the clients or at the edge of the brokers in the overlay looking for particular messages received or disseminated.

There are multiple testcases, each testing a single or combination of broker features (Table 6). Each of these testcases can be run separately or all in sequence by invoking the `AllTests` class. Test cases can be triggered with different broker/client configurations (Table 7). Each of these configurations is denoted by a *test version* number, which is specified at the command line when running the tests.

Test Case	Test For	Notes
`TestBroker`	basic broker operations such as correct initialization	tested with only one broker, no overlay.
`TestBrokerException`	throwing correct exceptions for single broker operation	tested with only one broker, no overlay
`TestClients`	client adv/pub/sub operations	tested with only one broker and multiple clients.
`TestBrokerException`	throwing correct exceptions for client operations	tested with one broker and two clients
`TestComplexCS`	similar to `TestClients` but with complex subscriptions	tested with only one broker and multiple clients.
`TestHeartBeat`	the heartbeat feature for detecting brokers	tested with two brokers in an overlay.
`TestMessageParser`	message parser to parse string-based pub/sub messages	component test, no brokers, no overlay

Table 6: PADRES testcases.

Irrespective of the testcase/test version being run, the tests must be run as a Junit test instead of a Java Application and its run configuration is set as if a broker is being run inside Eclipse (Section 5.1).

Most of the testcases uses the `MessageWatchAppender` to detect messages at various places. Generally an instance of this class is attached to the Logger so that whenever a message dissemination is logged, this message watcher can inform the testcase. The testcase will process the detected message to make proper assertions. A `PatternFilter` instance is used by the `MessageWatchAppender` to specify a specific set of messages to be detected.

Test Version	Descriptions
1	acyclic overlay; no adv/sub covering
2	acyclic overlay; no adv, but ACTIVE sub covering
3	acyclic overlay; no adv, but LAZY sub covering
4	acyclic overlay; adv covering ON; ACTIVE sub covering
5	FIXED cyclic routing; no adv/sub covering
6	FIXED cyclic routing; no adv, but ACTIVE sub covering
7	DYNAMIC cyclic routing; no adv/sub covering

Table 7: PADRES test versions.

16 Large Scale Deployment and Experimentation

16.1 Large-scale Deployment and Experimentation

PADRES is designed to run in a large-scale Internet sized environment. There is already undergoing work on a PlanetLab deployment and a tool for deploying and managing PADRES on PlanetLab nodes. More detailed information of these topics can be found in the PANDA section of the user guide.

16.2 Issues on Running PADRES in A Large-scale Internet Environment

When we run PADRES in a large network, for example, 25 brokers are running at the same time. We got some exceptions, which do not occur in a small network (e.g., several brokers). These exceptions are caused by not synchronizing variable correctly. We listed them below, and the solution as well.

ConcurrentModificationException in the broker side

```
Exception in thread "B1836" java.util.ConcurrentModificationException
       at java.util.HashMap$HashIterator.nextEntry(HashMap.java:787)
       at java.util.HashMap$KeyIterator.next(HashMap.java:823)
       at java.util.HashSet.writeObject(HashSet.java:254)
       at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
       at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
       at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
       at java.lang.reflect.Method.invoke(Method.java:585)
       at java.io.ObjectStreamClass.invokeWriteObject(ObjectStreamClass.java:890)
       at java.io.ObjectOutputStream.writeSerialData(ObjectOutputStream.java:1333)
       at java.io.ObjectOutputStream.writeOrdinaryObject(ObjectOutputStream.java:1284)
       at java.io.ObjectOutputStream.writeObject0(ObjectOutputStream.java:1073)
       at java.io.ObjectOutputStream.writeObject(ObjectOutputStream.java:291)
       at java.util.HashMap.writeObject(HashMap.java:985)
       at sun.reflect.GeneratedMethodAccessor7.invoke(Unknown Source)
       at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
       at java.lang.reflect.Method.invoke(Method.java:585)
       at java.io.ObjectStreamClass.invokeWriteObject(ObjectStreamClass.java:890)
       at java.io.ObjectOutputStream.writeSerialData(ObjectOutputStream.java:1333)
       at java.io.ObjectOutputStream.writeOrdinaryObject(ObjectOutputStream.java:1284)
       at java.io.ObjectOutputStream.writeObject0(ObjectOutputStream.java:1073)
       at java.io.ObjectOutputStream.defaultWriteFields(ObjectOutputStream.java:1369)
```

```
at java.io.ObjectOutputStream.writeSerialData(ObjectOutputStream.java:1341)
at java.io.ObjectOutputStream.writeOrdinaryObject(ObjectOutputStream.java:1284)
at java.io.ObjectOutputStream.writeObject0(ObjectOutputStream.java:1073)
at java.io.ObjectOutputStream.defaultWriteFields(ObjectOutputStream.java:1369)
at java.io.ObjectOutputStream.writeSerialData(ObjectOutputStream.java:1341)
at java.io.ObjectOutputStream.writeOrdinaryObject(ObjectOutputStream.java:1284)
at java.io.ObjectOutputStream.writeObject0(ObjectOutputStream.java:1073)
at java.io.ObjectOutputStream.writeObject(ObjectOutputStream.java:291)
at sun.rmi.server.UnicastRef.marshalValue(UnicastRef.java:258)
at sun.rmi.server.UnicastRef.invoke(UnicastRef.java:117)
at transport.RMITransportHandler_Stub.send(Unknown Source)
at binding.transportbinding.RMITransportBinding.handleMessage(RMITransportBinding.java:208
at brokercore.QueueHandler.run(QueueHandler.java:105)
```

SOLUTION: We found this exception usually occurs when one thread tries to modify a Collection while another thread is iterating over it. We noticed that the exception is thrown by RMITransportbinding. Moreover, Publications(Advertisements/Subscriptions) are routed between brokers and from broker to monitor/client, and there is a iterator in that toString() implementation. Because objects delivered using RMI must be serializable (implements Serializable). While in toString() methods we do use iterators, which may cause the ConcurrentModificationException. So we synchronized the hashMaps used in these toString() methods. (e.g., in the Advertisement.java) Add

```
synchronized(predicateMap)}
```

before

```
for (Iterator i = (predicateMap.keySet()).iterator(); i.hasNext();) {
        String attribute = (String) i.next();
        if (attribute.equalsIgnoreCase("class"))
         continue;
        Predicate p = (Predicate) predicateMap.get(attribute);
        stringRep += ",[" + attribute + "," + p.getOp() + ",";
        if ((p.getValue().getClass()).equals(String.class) || (p.getValue().getClass()).equals(Da
           stringRep += "\"" + p.getValue() + "\"";
        } else {
          stringRep += p.getValue();
        }
        stringRep += "]";
     }
```

We also need to make changes to advertisement/subscription/publication's Constructor. Synchronizing the HashMap at the creation time. (e.g., in the Advertisement.java) From:

```
public Advertisement() {
         advID = "";
         predicateMap = new HashMap();
        }
```

to:

```
public Advertisement() {
  advID = "";
         predicateMap = Collections.synchronizedMap(new HashMap());
        }
```

17 Open Issues

- Broker ID can not contain "_" (underscore) or "-" (hyphen) characters.
- NewRete matcher does not support composite subscription in the form of composite of composite events. e.g.:

  ```
  A || B && C || D
  ```

  It supports composite of composite and atomic events, like:

  ```
  A || B && C
  ```

18 Frequently Asked Questions

Also check the FAQ section of PADRES user guide.

19 Troubleshooting PADRES

This section describes PADRES build and runtime issues and the possible solutions.

19.1 PADRES Build-time Errors

Missing Parser Files

If you got the following problem:

```
The import ca.utoronto.msrg.padres.common.message.parser.XXX can not
be resolved.
```

You have to integrate JavaCC compiler to Eclipse. Please visit JavaCC Eclipse plugin homepage[5] to install JavaCC (use the update manager, if possible.) Please beware that the version 1.5.2 of JavaCC is not compatible with Eclipse 3.2.0, although JavaCC's website indicates it is so. If you are using Eclipse 3.2, the only way to get everything correct is to get the generated java files from someone or somewhere. Better, use the latest Eclipse release.

[5] http://eclipse-javacc.sourceforge.net/

19.2 PADRES Runtime Errors

In case of errors, try the following steps first:

- Try a clean rebuild (Building PADRES)
- Make sure command line options and environment variables are set correctly according to the "Running PADRES" guide.

Runtime error related to sh scripts under Cygwin

The following error

```
./startbroker: line 9: syntax error near unexpected token 'newline'
```

can be fixed by running the script file through dos2unix. The problem arises because of the difference in the new line representation in Windows and Unix OSes.

Runtime error related to log4j

The following error occurred because an old log4j.jar was in the PADRES build directory. It was linked instead of the new jar file. Removing the old jar fixed the problem.

```
ERROR transport.RMITransportHandler.<init>(RMITransportHandler.java:59) :
RMITransportHandler initialization failed: java.rmi.ServerError:
Error occurred in server thread; nested exception is:
java.lang.NoSuchFieldError: level
```

Runtime error related to postgres DB

In case of the following error, try checking the username and password for postgres DB to be correct.

```
Not Successful: An I/O error occurred while reading from backend -
Exception: java.net.SocketException: Connection reset
Stack Trace:
java.net.SocketException: Connection reset
at java.net.SocketInputStream.read(SocketInputStream.java:168)
at java.io.BufferedInputStream.fill(BufferedInputStream.java:218)
at java.io.BufferedInputStream.read(BufferedInputStream.java:235)
at org.postgresql.PG_Stream.ReceiveChar(PG_Stream.java:138)
at org.postgresql.jdbc1.AbstractJdbc1Connection.openConnection(AbstractJdbc1Connection.java:190
at org.postgresql.Driver.connect(Driver.java:122)
at java.sql.DriverManager.getConnection(DriverManager.java:525)
at java.sql.DriverManager.getConnection(DriverManager.java:171)
at binding.dbbinding.DBConnector.startup(DBConnector.java:151)
at binding.dbbinding.DBBinding.init(DBBinding.java:61)
at binding.dbbinding.DBBinding.<init>(DBBinding.java:53)
at controller.LifeCycleManager.<init>(LifeCycleManager.java:85)
at controller.Controller.run(Controller.java:73)
End of Stack Trace
```

Runtime error related to X11 Window

The following error is caused when the display device is lacking. Please refer the Headless Mode in the Java SE Platform to fix it.

```
java.lang.InternalError: Can't connect to X11 window server
using ':0.0' as the value of the DISPLAY variable.
at sun.awt.X11GraphicsEnvironment.initDisplay(Native Method)
at sun.awt.X11GraphicsEnvironment.<clinit>(X11GraphicsEnvironment.java:134)
at java.lang.Class.forName0(Native Method)
at java.lang.Class.forName(Class.java:141)
at java.awt.GraphicsEnvironment.getLocalGraphicsEnvironment(GraphicsEnvironment.java:62)
at sun.awt.motif.MToolkit.<clinit>(MToolkit.java:81)
at java.lang.Class.forName0(Native Method)
at java.lang.Class.forName(Class.java:141)
at java.awt.Toolkit$2.run(Toolkit.java:748)
at java.security.AccessController.doPrivileged(Native Method)
at java.awt.Toolkit.getDefaultToolkit(Toolkit.java:739)
at java.awt.Toolkit.getEventQueue(Toolkit.java:1519)
at java.awt.EventQueue.invokeLater(EventQueue.java:792)
at javax.swing.SwingUtilities.invokeLater(SwingUtilities.java:1170)
at javax.swing.Timer.post(Timer.java:538)
at javax.swing.TimerQueue.postExpiredTimers(TimerQueue.java:193)
at -D(TimerQueue.java:229)
at java.lang.Thread.run(Thread.java:534)
```

You have to use the following command line to run the application.

```
java -Djava.awt.headless=true
```