Presentation of the TOTEM Communication Infrastructure

Denis Conan, Michel Simatic and Gabriel Adgeg

March 7, 2012

History of changes

Date	**Changes**
31st January, 2011	Initiate with the first version.
4th February, 2011	Make it visible to TOTEM partners. Not finalized to be studied and needs proof reading. Contribution is mainly in the appendices.
18th February, 2011	New Section 5 “Example application demonstrating the integration of the technologies” (Django, RabbitMQ Broker, RabbitMQ Python Client with Pika, RabbitMQ Java Client, RabbitMQ Java Client for Android), game room completed and game play partly specified. Move from Kombu to Pika.
21th February, 2011	Small corrections. For testing the example application, follow the instructions of Appendix B (except B.2) and Section 5.3
25th February, 2011	Section 5.2.3 “Communication architecture during the game play” complete. Code of the example application demonstrating game room and game play updated.
15th March, 2011	First snapshot version of the Java library for using RabbitMQ in TOTEM in the applications GameMaster, Spectator, and Player. This API is to be described in a dedicated section of the document.
21th March, 2011	First snapshot version of the Python library for using RabbitMQ in TOTEM in the GameServer (GameRoom server and ApplicationInstance server). This API is to be described in a dedicated section of the document.
23th March, 2011	“Application instance” is preferred to “game play”. The Section A of the appendix describing EBS, AMQP and RabbitMQ is completed.
28th March, 2011	New Section on the “API of the communication infrastructure”, that is on the user manual of the communication infrastructure using RabbitMQ. Java Android TOTEM library decomposed into four modules: common, gamemasterapplication, playerapplication, and spectatorapplication.
29th March, 2011	New Section on the “Requirements”, with for each of them some comments and the status. Add status of sections in the introduction. First draft of the Section 2 introducing the illustrative application. The content was previously in the analysis subsection of Section 10. First draft of the Section 4 introducing the AMQP concepts. The content was previously in the appendix. Example integration application with Java Android Clients, Game Master, Player, and Spectator.

30th of March, 2011	Add a figure of AMQP concepts in Section 4. First draft of the system architecture with a focus in communication infrastructure in Section 5. First draft of the detailed design, with figures moved from Section 10 to Section 6.
15th of April, 2011	Moving RabbitMQ server from version 2.3.1 to version 2.4.1. Reorganize Section A and add a subsection on a JavaScript client example.
30th of April, 2011	The Section "Requirements" has been updated to current status. The game room server using RabbitMQ is replaced by a game server using XML-RPC to control the game instances, which use RabbitMQ. The term "application instance server" is replaced by "game logic server" and the code is separated from the game server. The API is complemented with methods to publish to all and to a consumer. New section "Organisation of the source code" to describe the directory structure of the Subversion Repository.
2nd of May, 2011	Update of the Section "Example application demonstrating the integration of the technologies". Android clients updated to the last version of the example application.
9th of May, 2011	Add instructions in Section "Installation of RabbitMQ", for installing and checking the version number of Erlang, Sun-JDK, Maven, and Python. Add instructions in Section "Installation of RabbitMQ" for w-getting RabbitMQ. Add a new Section "Short tutorial on how to design and implement a new protocol". The example application does no more use Django, the integration to the Django game server is done elsewhere. Any previously existing Django parts are removed.
11th of May, 2011	Defensive programming of the Java TOTEM Communication Library: the semantics of the publications is "at least once", thus the Section "Detailed design of the communication infrastructure" is updated with this new design decision. New script for the automatic running of the example application with Android Emulators.
26th of May, 2011	In order to prepare the integration with the MPEG4 Player, the TOTEM library is complemented with a TOTEM RabbitMQ Proxy on Android.
8th of July, 2011	Appendix updated with version 2.5.1 of RabbitMQ. New Author Gabriel Adgeg.
2nd of August, 2011	New appendix for RabbitMQ client in Javascript with node.js. Former Javascript appendix is not removed, since it should be used with clients that cannot use node.js.
9th of August, 2011	Corrections added to the appendix for RabbitMQ client in Javascript with node.js. Useless diagram removed. Output of the execution of the Step1 added.

22nd of August, 2011	New project in directory `RabbitMQ/Tests/proxy-node-amqp`. A proxy implemented with Node.js is provided to communicate with the RabbitMQ broker. A web client is also provided to communicate with the proxy in order to receive messages from a given queue. All the details to install and launch this example are located in the file `readme.txt`.
27th of September, 2011	New version of Android Java client applications of the integration example application.
17th of October, 2011	Move to version 2.5.1 of RabbitMQ. Deprecated appendix for RabbitMQ Javascript plugin removed. Appendix for Java J2SE clients is set to deprecated since these have not been updated to the latest version of TOTEM library. Description of Javascript subsystems for Master and Spectator applications running in a browser added. Installation procedure of XMLRPC library for Node.js added.
18th of October, 2011	Appendix for installing RabbitMQ on Amazon EC2 stated to status deprecated since the section has not already been updated to the last version. Appendix for adapting RabbitMQ Java client library to Android phones removed since the adaptation is not needed anymore.
25th of October, 2011	Implementation and execution note updated with new Javascript configuration files. Node.js installation procedure corrected. References to RabbitMQ proxy for MPEG player removed.
9th of November, 2011	New figures to present the architecture of the middleware. Notes added on parts that will be updated soon. Organisation of the source code updated.
7th of December, 2011	Python source code modified to fix the pika.NotImplementedError bug.
15th of December, 2011	Full support of several parallel game instances. New XMLRPC methods to list game instances or to terminate a single game instance. RabbitMQ-Totem library in JavaScript: state machine, action kind, actions, transparent consumption of messages. Possibility to host several Masters and Spectators JavaScript applications of different games on a single NodeJs proxy. Python projects for GameServer and GameLogicServer changed in PyDev projects.
22th of December, 2011	Performances of the middlaware are added to the requirements section.
18th of January, 2012	Installation of NPM updated, since curl command is required. Version 0.1.0 of AMQP library for Node is required, since bugs seem to appear in later versions.

6th of February, 2012	References to Logging server changed in Chapter 2. Requirements updated with precisions about the presence service and HTTP communication. Short presentation of Node.js, and of its choice in Chapter 4. Design decisions concerning AMQP ressources added in Chapter 6.
6th of February, 2012	Presentation of AMQP concepts for JavaScript in Chapter 6. New XML-RPC methods added in the Figure of the architecture-game-server. JavaScript idioms added to Chapter 7. JavaScript API directory is added to Chapter 8. Tutorial replaced with the one used during the Lab. Some explanations have been modified for a better understanding.
15th of February, 2012	Presentation of the concepts of the middleware(the use of XML-RPC, actions and actions kinds...) added to the tutorial Section. Many details added to the tutorial to detail the points where students had problems during the Lab. New tutorial for the JavaScript applications. Methods refactored with new version of API in the tutorial. Section "Integration of technologies" updated to new contents on the SVN Obsolete appendix for Android phones removed. Appendix for J2SE applications corrected. Version of the RabbitMQ broker is set to 2.7.1.

Contents

License of TOTEM communication middleware

1 Introduction

NB: This document is a living document.

Status of the sections of the document:

- Section 1, “Introduction”: Complete.
- Section 2, “Illustrative scenarios and use cases”: Complete.
- Section 3, “Requirements for the communication infrastructure”: Complete.
- Section 4, “Presentation of EBS, AMQP, RabbitMQ and Node.js”: Complete.
- Section 5, “Architecture of the communication infrastructure”: Complete.
- Section 6, “Detailed design of the communication infrastructure”: Complete.
- Section 7, “API of the communication infrastructure”: Complete.
- Section 8, “Organisation of the source code”: Complete.
- Section 9, “Tutorial on how to design and implement a new protocol”: Complete.
- Section 10, “Example application demonstrating the integration of the technologies”: Complete.
- Section A, “Installation of RabbitMQ”: Complete.
- Section B, “Installation of RabbitMQ and Pika on Amazon EC2”: Complete but deprecated.

2 Illustrative scenarios and use cases

In this section, and through the document, we illustrate the TOTEM communication infrastructure with an illustrative application demonstrating the integration of many of the TOTEM technologies. The objective of this example application is to help in eliciting the requirements of the communication infrastructure in Section 3. The subsystems and the actors are modelled in the UML use case diagrams of Figures 1 and 2. Since this document describes the communication infrastructure, we decided to present two different diagrams to better state where and when the two technologies XML-RPC and RabbitMQ are involved: Figure 1 shows the subsystems at the beginning of the application when the end-users log in using the XML-RPC communication technology; Figure 2 shows the subsystems during the game play of a game instance when the communication exchanges are performed using the RabbitMQ technology [1]:

- The actors of the use case diagrams of Figures 1 and 2 are the participants of the example application and the systems are the subsystems. In Figure 2, since the Game Broker is not a "functional" but a "technical" subsystem, it is not present in the use case diagram. Please observe also that we do not restrict ourselves and allow for direct point to point communication between the Master Application, the Player Application and the Spectator System, without the intermediary of the Game Server. All the relevant events are logged by the Logging Server, for instance for post-mortem debugging. It provides logging functionalities to all the other subsystems (this is modelled in Figure 2 by the many arrows pointing to the LoggingServer).
- End-user subsystems connect to the Game Server and log in. Game instances are created by the Game Server. For the sake of simplicity, we consider only one game server managing all the games and all the game instances. In this document, we consider only one game instance. A game instance is managed by a Game Logic Server, which is the server that contains all the game logic of the game for that instance.
- The Master Application is the application used by the game master to create and manage a new game instance. For the sake of simplicity, we only assign a name to a game and a name to a game instance. In the example application, game masters do not intervene to accept or refuse the joining of players or spectators.
- The Player Application corresponds to the game application executed by the game player. In this simplistic example, a player application can only join or leave a game instance, and exchange notifications to and from the game server or the other game entities (master, players, spectators).
- The Spectator System is used by non-players to be informed about events of the game instance. Spectators register to categories of events. The events sent to the spectators do not need to pass through the Game Logic Server

[1]For the sake of simplicity, we have ignored the termination of the application bringing another subsystem into play, which calls the Game Server for organising the termination of all the subsystems using AMQP messages.

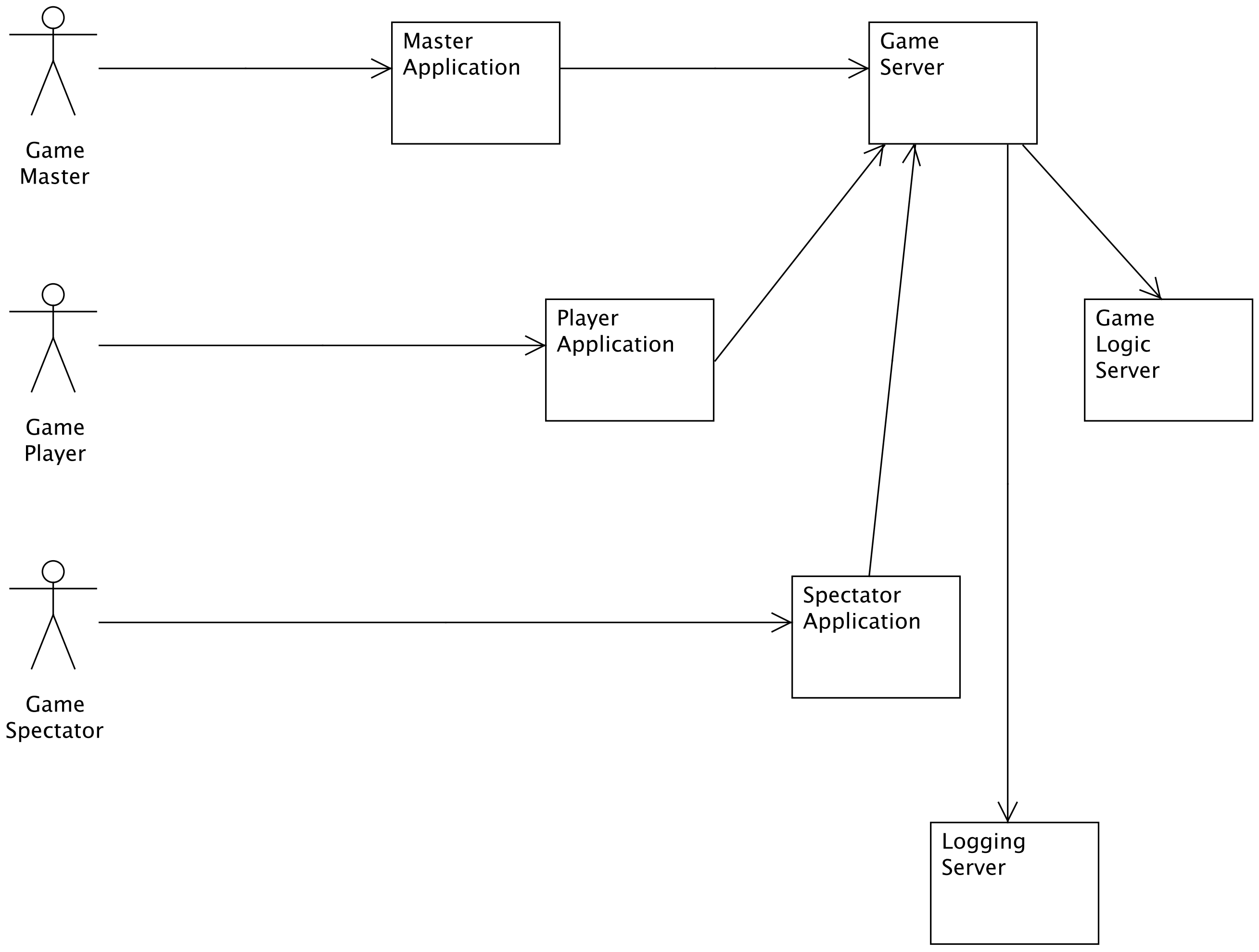

Figure 1: Illustrative use case —when log in using XML-RPC

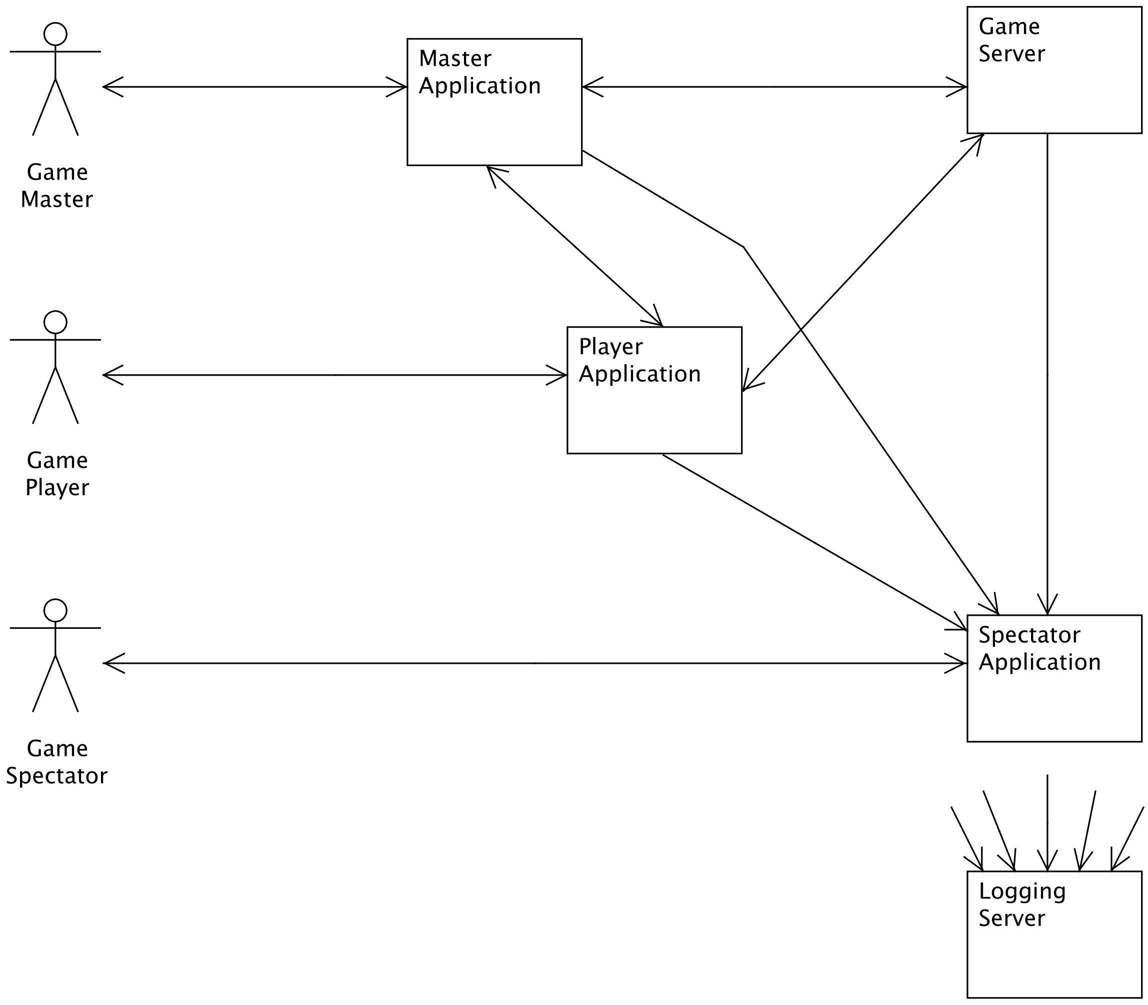

Figure 2: Illustrative use case —during the game play usingRabbitMQ

3 Requirements for the communication infrastructure

The requirements listed in this section are the requirements for the communication infrastructure with their corresponding status.

1. Messaging
   - (a) Destination of message
     - i. Point-to-point: From one entity (game server, game master, gamer / player, or spectator) to any other entity which the sending entity is aware of. (Mandatory)
       - Done ▶All the bindings (between exchanges and queues) and the routing keys of the messages include the identity of the sender and of the addressee or "all" (all the entities).◀
     - ii. Broadcast to game players: It is possible for one entity (game server, game master, gamer / player, or spectator) to broadcast to all the other entities (including itself) (Mandatory)
       - Done ▶Using "all" as the addressee in the routing key, a message is broadcast to all the entities of the game instance. Note that the sender also receives its broadcast message.◀
   - (b) Type of messages
     - i. Notification: An entity is able to send ("fire and forget" mode) an information to other entity(ies). (Mandatory)
       - Done ▶This is the messaging paradigm of AMQP[2].◀
     - ii. Request/Response: When an entity sends such a message, it blocks itself until it receives a response to its request (RPC mode). (Mandatory)
       - Done ▶This is the messaging paradigm of XML-RPC. The login exchanges are performed using RPC mode.◀
     - iii. Request/Response with "future": When an entity sends such a message, it continues operation and receives a delivery receipt at a later stage. (Important)
       - Done ▶Futures can be implemented during the game play. This feature is not demonstrated in the illustrative application. But ask for an example to communication infrastructure designers when needed.◀
   - (c) Message reliability: Each message has a "reliability" attribute. (Important)
     - If this attribute is set to true, message will arrive to all of its destinations (even in the case of a broadcast message), even though some mobiles crashed or are disconnected. However, if a mobile crashes and recovers, or disconnects for more than a given number of minutes (this number is configured at game server level), messages will be lost (this is to prevent the broker and the mobiles from being overflowed by undelivered messages).
       - Done ▶This corresponds to the "persistent" attribute of messages in AMQP specification. By default, messages are not persistent. This feature is not demonstrated in the illustrative application. But ask for an example to communication infrastructure designers when needed. In addition, AMQP queues can be durable, that is they survive a broker crash and restart.◀
     - If this attribute is set to false, messages can be lost.
       - Done ▶This is the default configuration of AMQP messages.◀
   - (d) Order of messages

[2]The middleware RabbitMQ used for communication during the game play conforms to the standard AMQP, Advanced Message Queuing Protocol. Both AMQP and RabbitMQ are introduced in Section 4.

   i. FIFO: Messages originating from the same source are received in the order with which they were sent by all the recipients. (Mandatory)

      - **Done** ▶RabbitMQ guarantees ordering of the messages, as long as the messages in question follow the exact same path through the clients and the broker (same publisher, same exchange, same queue and same consumer). Be careful! This remains true as long as the broker does not crash and recover.◀

(e) Platform-independent message encoding: The communication middleware is able to take care of problems related to differences of data encoding on the different platforms participating to the communication (Endian problem, Strings encoding...). (Important)

   - **Done** ▶Rabbit treats a message payload as an opaque binary. We use Strings coding.◀

(f) Message security: Messages can be encrypted in order to avoid the decoding of messages contents by third parties. (Nice to have)

   - **OpenIssue** ▶AMQP promotes the use of SASL, which hasn't been experimented yet.◀

2. Presence service: At any time, an entity (mobile or game server) knows which session members are accessible.

   - **Postponed**

3. Disconnection management (Mandatory, risk Medium). In case of disconnection (triggered by the application or not) from the broker, a mobile (Reminder: we assume that the game server is always connected to the broker) must:

   (a) Be aware of disconnection from network

      - **Done** ▶Libraries developed for Java J2SE, Android and JavaScript applications have a "maximum number of retries" property, which can be set by the user. If a disconnection occurs during the consumption or the publication of messages, the library tries to reconnect, to consume and possibly to publish messages (every second) until the maximum number of retries value is reached. If the network stays unreachable after this time period, the application stops its reconnection routine and the messages intended to this application stay on the broker queues.◀

   (b) Be aware of the disconnection from the broker

      - **Postponed**

   (c) Have tools to go on communicating:

      i. The communication middleware always accepts messages from the application which uses it, even if it cannot communicate with the other entities (e.g. other mobiles and server) participating to the protocol.
      Said in other words: Messages are queued locally on the mobile rather than being deleted because the mobile is not connected to the broker. When connection to the broker is back, these queued messages are sent.

         - **Postponed**

      ii. Each message can be given a delivery timeout. If such a timeout is set and the message is not delivered to destination after such timeout, an exception is sent to the application. The application can invoke a primitive to cancel such a message.

         - **Postponed**

4. Can send small messages as well as large binary blobs (e.g. from Location Survey tool). (Important)

- Done ▶By introducing the concept of framing, AMQP messages are chopped up into frames for interleaving on transport, which is useful for instance for large messages.◀

5. Totem offers game session concept above the communication middleware. In particular, broadcasts are sent to all session members. (Important)

   - Done ▶This is the role of this document to introduce for such concept of game session through the AMQP concepts of virtual host, exchange, queue, routing key and binding.◀

6. Integrates with web framework

   (a) The web framework is able to send/receive messages through/from the communication middleware. (Mandatory)

   - Done

   (b) The code of the broker can be integrated within the code of the framework. (Nice to have)

   - Done

7. An (Android) mobile client is able to send/receive messages through/from the communication middleware. (Mandatory)

   - Done ▶The integration of Android running mobile devices is demonstrated in the example application of Section 10.◀

8. A client (mobile or server) can communicate with the communication middleware through HTTP protocol. This protocol is used to connect the MPEG player and the communication middleware. This protocol may also be used in the case Telecom operators do not allow direct socket connections from mobile to the Internet. In this case, the mobile has to use HTTP protocol instead of plain socket-based library. (Mandatory)

   - Done ▶Libraries are provided to enable communication between JavaScript applications and the broker (through a proxy). The communication is made with HTTP via AJAX requests.◀

9. Easy setup and deployment. (Mandatory)

   - Done ▶This document includes several sections in the appendix describing installation procedures.◀

10. Standards based (Important).

    - Done ▶RabbitMQ conforms to the version 0.9.1 of AMQP standard.◀

11. Broker is able to push messages to game server (Mandatory, risk zero)

    - Done ▶AMQP protocol pushes messages as soon as the consumer asks for it. By dedicating a thread for consuming messages in a loop, messages will be pushed by the broker at the speed of consumer calls of the AMQP command `consume`.◀

12. Broker is able to push messages to mobiles (Important as it lowers latency for delivering messages to mobiles. Nevertheless it is not mandatory, as game design can limit the visibility of an important latency.)

    - Done ▶Same answer as for Requirement 11.◀

13. On the mobile, if multiple communication channels (Wifi, GPRS, etc.) are available

(a) When the player launches a game, the game tries to connect to the broker: We rely on the mobile OS to choose (or give to the user the choice of) the communication channel to use to connect to the broker (Mandatory, risk Medium: On Android, we can use method setNetworkPreference of the ConnectivityManager but there is little documentation).

- Irrelevant ▶Not an issue for the communication middleware.◀

(b) In the case the mobile was connected to Wifi and the player gets out of the Wifi zone, the telephone reconnects automatically to another available network (Nice to have because the player is not aware that it is going to use her data plan now).

- Irrelevant ▶Not an issue for the communication middleware.◀

(c) We keep track of "free" Internet connections of the user

- Irrelevant ▶Not an issue for the communication middleware.◀

14. Integrates with Android Cloud to Device Messaging Framework (Nice to have, as it is Android specific.)

- Irrelevant ▶It is a feature which allows to send small out of application messages. It does not have to be integrated with RabbitMQ.◀

15. Latency. Communication middleware requires a maximum of 100 milliseconds (100 is an estimation which should fit most of Totem games) to send a message from a mobile client to the game server (if the mobile is connected with the broker thanks to a Wifi network).

- Done ▶In the following array, we present the average performances for the publication and the consumption of one message containing about 10 characters. Results are given in milliseconds.

Client	Network	Publish	Consume
Android application	WiFi	2.9	7.8
	3G+	1.6	120
JavaScript application	WiFi	10.9	111
	3G+	18.2	371

◀

16. Throughput. Communication middleware is able to withstand at least 1 notification per mobile client per second per game server (the configuration of each game server is the configuration of a free Amazon server), with at least 500 mobile clients. Above 500 clients (this 500 number is currently an estimation; A more precise value should be computed thanks to a marketing study!), we consider that the game editor has enough customers to be able to pay for the Cloud service.

- OpenIssue ▶Confident, but not experimented because no performance tests, yet.◀

17. Identity, user management (ideally via Facebook Connect, OAuth, OpenID, etc.) (Mandatory). If the communication middleware requires user login functionality, this functionality database can be replaced by mechanisms like Facebook Connect, OAuth, OpenID, etc.

- Irrelevant ▶The TOTEM Django game server manages the log-ins and passwords. Thus, it is not an issue for the communication middleware.◀

18. Cloud support: Amazon, or Google App Engine. (Mandatory)

- Done ▶The cloud integration is not demonstrated with Google App. Engine (since the latter provides its own software infrastructure), but with Amazon EC2. Cf. Section B of the appendix for the installation procedure.◀

4 Presentation of EBS, AMQP, RabbitMQ and Node.js

This section presents the communication infrastructure concepts of the AMQP specification and the middleware RabbitMQ used for event-based notification in the TOTEM project. The other communication paradigm used in the TOTEM project is RPC, more precisely by bringing into play the XML-RPC standard. Since TOTEM partners are more used with XML-RPC, we only present AMQP.

4.1 Presentation of Event-Based Systems

In this section, we briefly introduce the basic concepts and terminology of event-based systems. According to the terminology of event-based systems [Mühl et al., 2006], an "event" is any happening of interest that can be observed from and within a computer. Event examples are physical event, timer event, etc. A "notification" is an object[3] that contains data describing the event. A "producer" is a component[4] that publishes notifications. A "consumer" is a component that reacts to notifications delivered to it by the notification service. A "subscription" describes a set of notifications a consumer is interested in.

Figure 3 compares the model of interaction of event-based systems with other well-known architectures, namely RPC, callback and anonymous RPC [Mühl et al., 2006]:

- Request/reply: the consumer requests some data from the provider whose identity is known by the consumer.
- Callback: the consumer registers at a specific provider its interest to be notified whenever some condition becomes true; the identity of the components is known and must be managed on both sides.
- Anonymous request/reply: the consumer does not know the identity of provider(s), one request may result in an unknown number of replies.
- Event-based: the producers of notifications initiate the communication. Producers do not know any consumers. If a notification matches a subscription, it is delivered to the registered consumers.

		Initiator	
		Consumer	Provider
Addressee	Direct	Request/Reply	Callback
	Indirect	Anonymous Request/Reply	Event-Based

Figure 3: Models of interaction [Mühl et al., 2006]: "initiator" describes whether the consumer or the provider initiates the interaction; "addressing" indicates whether the addressee of the interaction is known or unknown. The tradeoff is between the simplicity of request/reply and the flexibility of event-based interaction.

The generic architecture of a distributed event-based system is depicted in Figure 4. The notification service forms an overlay network in the underlying system. The overlay consists of event brokers that run as processes on physical nodes. Local brokers put the first message into the network. Border and inner brokers forward the message to neighboring brokers according

[3]The term "object" is employed here in its general meaning, not in the sense it possesses in object-oriented software engineering.

[4]The term "component" is employed here in its general meaning, not in the sense it possesses in component-based software engineering.

to filter-based routing tables and routing strategies. Messages are sent to local brokers. Local brokers deliver the message to the application components.

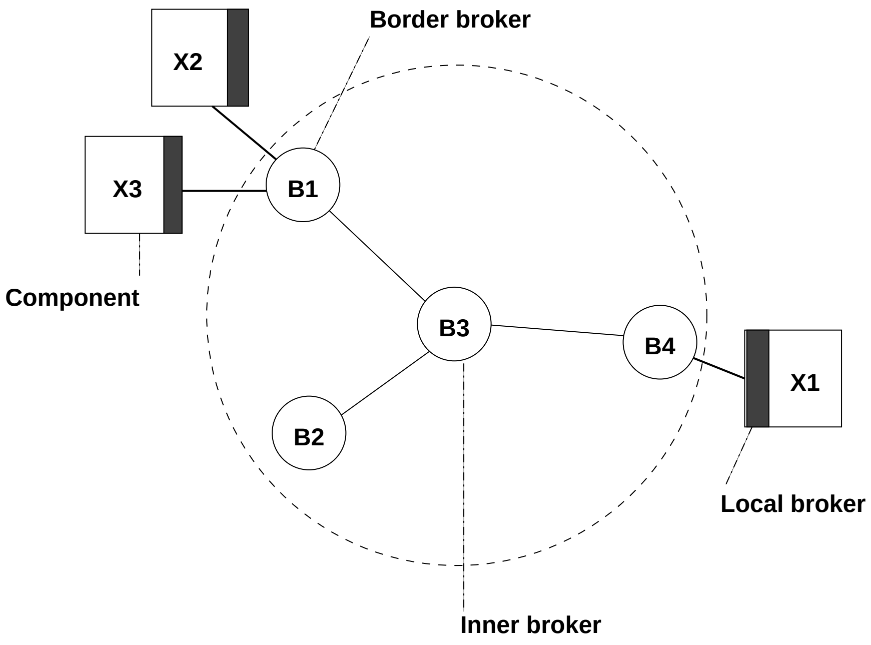

Figure 4: Generic architecture of a distributed event-based system

Event-based systems are classified according to their notification filtering mechanisms. Four categories of notification filtering mechanisms exist:

- In channel-based filtering, producers select a channel into which a notification is published, and consumers select a channel and will get all notifications published therein. Channel identifier is only the visible message part to the event-based service. An example of this mechanism is the Corba Event Service.

- In subject-based filtering, consumers use string matching for notification selection. Each notification and subscription is defined as a rooted path in a tree of subjects. For instance, a game instance exchange application publishes new events of GameInstanceOne under the subject Game.Instance.GameInstanceOne.ActionOne and consumers subscribe for Game.Instance.* to get all the events that concerns game instances.

- In type-based filtering, consumers use path expressions and subtype inclusion to select notifications.

- In content-based filtering, filters are evaluated on the whole content of notifications (data and meta-data). Message delivery is based on a query or predicate issued by the subscriber. Available solutions are template matching, extensible filter expressions on name value pairs, XPath expressions on XML, etc.

Distributed notification routing[5]:

- Routing is the functionality of matching all notifications with all subscriptions, and of delivering notifications to all clients and neighboring brokers with a matching subscription.

[5]This paragraph is added for the sake of completion since the AMQP standard does not define distribution notification routing functionalities.

- The first strategy is “flooding”: Brokers forward notifications to all neighboring brokers; Only brokers to which subscribers are connected test on matching subscriptions. The main advantage of flooding is that it guarantees that all the notifications will reach their destination. The main drawback of flooding is that many unnecessary messages are exchanged among brokers.
- The other strategy, called “filter-based”, depends on routing tables which are maintained by brokers. A routing entry is a filter-destination pair (F, D). Entries are updated by sending control messages.

4.2 Presentation of Advanced Message Queuing Protocol

4.2.1 Introduction to AMQP

This section is a collection of relevant excerpts from the public Web site of the Consortium `https://www.amqp.org`.

AMQP is an open standard for messaging middleware. The initial version (0.8) was released by the AMQP Working Group as a Published Specification in June 2006. In December 2006, V0-9, V0-91 and V0-10 were also released as a Published Specifications. AMQP1.0 will be licensed in the same way as previous versions but no guarantee of backwards compatibility can be given prior to version 1.0.

JMS is an API. HTTP is a protocol. AMQP delivers the middleware equivalent of HTTP while leaving it up to others to provide implementations. JMS does not specify the implementation or the wire-level protocol. JMS is not technology agnostic and only legitimately supports Java platforms under the terms of its licensing (there will be a product which provides a JMS interface, but a JMS-like interface cannot legally be provided for non-Java platforms). AMQP provides a superset of the semantics required to implement JMS, but also enables APIs for C, C++, Python, C# or any other language on Linux, Solaris, Windows, Z/OS, etc. The AMQP Working Group is not initially focusing on standardizing an API for AMQP implementations. It will be natural for programming environments to create API’s onto AMQP which are natural for programmers of that environment; an API for Java is likely to look like JMS but an API for Python or COBOL may look quite different. Despite being written to different API’s, implementations which are AMQP compliant will inter-operate seamlessly; so a Java program could use an AMQP compliant JMS to communicate with a .NET program which is using a different API. That’s an advantage of wire-level protocols.

CORBA/IIOP is a wire-level protocol for remote object invocation. You get a handle on an object and call a method. This is different from DCOM, where you get a handle on an “interface”, and ONC/RPC and DCE, where you get a handle on a process. All of these are synchronous networked calls, and there is no notion of guarantee, or queuing, and little notion of QoS. Protocols like these have a place but they are incomplete on their own. IIOP also doesn’t play nice with firewalls, which crop up frequently in real application scenarios. AMQP is conceptually similar to the CORBA Notification Service and CORBA Event Service. However, there are few implementations of the Notification or Event services in part because of the complexity of the specifications and you need a full ORB to run it. This complexity is not amenable to wide spread adoption.

4.2.2 AMQP concepts

This section is adapted from excerpts of the specification [AMQP Consortium, 2008]. The concepts are depicted in Figure 5. First of all, notice that AMQP is mainly one-broker-centered and that the specification ignores distributed notification routing. Of course, implementations of the standard are not prevented to provide such functionalities. For instance, RabbitMQ allows exhange-to-exchange bindings and a “shovel” binding for transferring messages from one node to other nodes in a cluster of hosts.

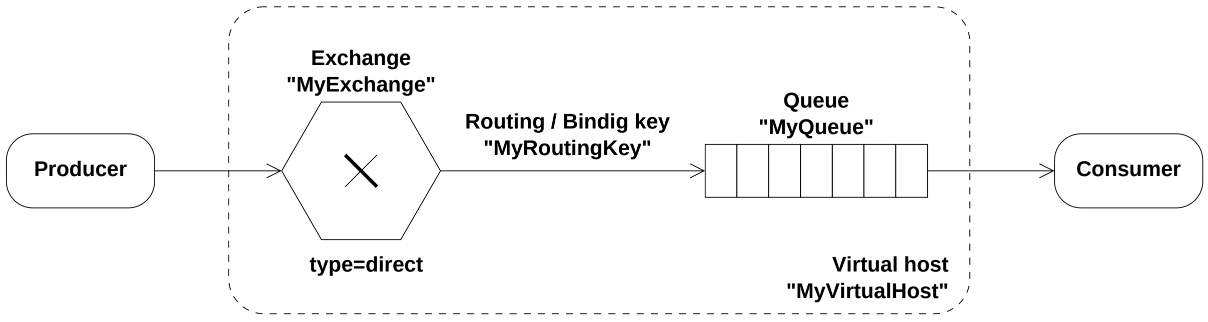
Producer
Exchange "MyExchange"
type=direct
Routing / Bindig key "MyRoutingKey"
Queue "MyQueue"
Virtual host "MyVirtualHost"
Consumer

Figure 5: AMQP concepts

The AMQ model consists of a set of components that route and store messages within the broker service, plus a set of rules for wiring these components together. The AMQ model specifies a modular set of components and standard rules for connecting these. There are three main types of component, which are connected into processing chains in the broker to create the desired functionality:

- The *exchange* receives messages from publishers and routes these to “message queues”, based on arbitrary criteria, usually message properties or content.
- The *message queue* stores messages until they can be safely processed by a consumer (or multiple consumers).
- The *binding* defines the relationship between a message queue and an exchange, and provides the message routing criteria.

Using this model, it is possible to emulate the classic message-oriented middleware concepts of store-and-forward queues and topic subscriptions.

A virtual host is a data partition within the broker, it is an administrative convenience which will prove useful to those wishing to provide AMQP as a service on a shared infrastructure. A virtual host comprises its own name space, a set of exchanges, message queues, and all associated objects. Each connection must be associated with a single virtual host. All channels within the connection work with the same virtual host. There is no way to communicate with a different virtual host on the same connection, nor is there any way to switch to a different virtual host without tearing down the connection and beginning afresh. The protocol offers no mechanisms for creating or configuring virtual hosts —*i.e.*, this is done in an undefined manner within the broker and is entirely implementation-dependent. The authentication scheme of the broker is shared between all virtual hosts on a broker. However, the authorization scheme used may be unique to each virtual host. Administrators who need different authentication schemes for each virtual host should use separate brokers.

The concept of virtual host is essential to TOTEM since it allows for a complete isolation between the different application instances.

Message Queue. A message queue stores messages in memory or on disk, and delivers these in FIFO sequence to one or more consumers. Message queues are message storage and distribution entities. They are created and maintained by the broker and the broker architecture. A message queue has various properties: private or shared, durable or temporary, client-named or broker-named, etc.

In the presence of multiple readers from a queue, or client transactions, or use of priority fields, or use of message selectors, or implementation-specific delivery optimizations, the queue may not exhibit true FIFO characteristics. The only way to guarantee FIFO is to have just one consumer connected to a queue.

Message queues may be durable, temporary, or auto-deleted. Durable message queues last until they are deleted. Temporary message queues last until the broker shuts-down. Auto-deleted message queues last until they are no longer used.

By selecting the desired properties, one can use a message queue to implement conventional middleware entities such as:

- A shared store-and-forward queue, which holds messages and distributes these between consumers on a round-robin basis. Store and forward queues are typically durable and shared between multiple consumers.

- A private reply queue, which holds messages and forwards these to a single consumer. Reply queues are typically temporary, broker-named, and private to one consumer.

- A private subscription queue, which holds messages collected from various “subscribed” sources, and forwards these to a single consumer. Subscription queues are typically temporary, broker-named, and private to one consumer.

These categories are not formally defined in AMQP: they are examples of how message queues can be used.

An acknowledgment is a formal signal from the consumer to a message queue that it has successfully processed a message. There are two possible acknowledgment models:

1. Automatic, in which the broker removes a content from a message queue as soon as it delivers it to a consumer.

2. Explicit, in which the consumer must send an `Ack` method for each message, or batch of messages, that it has processed. The client layers can themselves implement explicit acknowledgments in different ways, *e.g.* as soon as a message is received, or when the application indicates that it has processed it. These differences do not affect AMQP or interoperability.

Most of the time, consumers rely on the automatic acknowledgment mechanism. Since no specific QoS requirements are expressed, this is also our choice for TOTEM.

Exchange. An exchange accepts messages from a producer and routes these to message queues according to pre-arranged criteria. These criteria are called “bindings”. Exchanges are matching and routing engines. That is, they inspect messages and using their binding tables, decide how to forward these messages to message queues or other exchanges. Exchanges never store messages.

A binding is a relationship between a message queue and an exchange. The lifespan of bindings depend on the message queues they are defined for —*i.e.*, when a message queue is destroyed, its bindings are also destroyed.

AMQP defines a number of standard exchange types, which cover the fundamental types of routing needed to do common message delivery. Exchange types are named so that applications which create their own exchanges can tell the broker what exchange type to use. Exchange instances are also named so that applications can specify how to bind queues and publish messages.

Routing key and binding key. In the general case, an exchange examines a message's properties, its header fields, and its body content, and using this and possibly data from other sources, decides how to route the message. In the majority of simple cases, the exchange examines a single key field, which is called the "routing key". A *routing key* is a virtual address that the exchange may use to decide how to route the message. For point-to-point routing, the routing key is usually the name of a message queue. For topic pub-sub routing, the routing key is usually the topic hierarchy value. In more complex cases, the routing key may be combined with routing on message header fields and/or its content.

Routing keys are used by producers in messages to indicate the "address" of message consumers. When binding a message queue to an exchange, another key is given, called the *binding key*. This latter key is the parameter used by the exchange to configure the routing protocol implemented by the exchange. Therefore, the routing algorithm uses the routing key of a message and the binding key of the bound message queues to route the message to consumers attached to message queues. Note that most of the time, the terms "routing key" and "binding key" are used interchangeably.

Notification filtering mechanisms. In the implementations of the AMQP specification we are aware of, the routing key is the only part of the header used for routing, that is the message content is not used for routing; then, "content-based filtering" is allowed in AMQP but for instance not available in the RabbitMQ implementation. In addition, messages are arrays of bytes and are not objects, in the object oriented acceptation; then, "type-based filtering" is not part of AMQP. Therefore, the notification filtering mechanisms available are "channel-based filtering" and "subject-based filtering". The notification filtering mechanisms are specified via the exchange types.

Exchange types. AMQP specify three main types of echanges. In the TOTEM project, we use the most powerful type, namely the topic exchange type that implements subject-based filtering.

The *direct exchange type* implements a simplistic form of subject-based filtering and works as follows:

1. A message queue binds to the exchange using a routing key K. Message queues can bind using any valid routing key value, but most often message queues will bind using their own name as routing key.
2. A publisher sends the exchange a message with the routing key R.
3. The message is passed to the message queue if $K = R$.

The *fanout exchange type* implements channel-based filtering and works as follows:

1. A message queue binds to the exchange with no arguments.
2. A publisher sends the exchange a message.
3. The message is passed to the message queue unconditionally.

The *topic exchange type* works as follows:

1. A message queue binds to the exchange using a binding key B as the routing pattern.
2. A publisher sends the exchange a message with the routing key R.
3. The message is passed to the message queue if R matches B.

The routing key used for a topic exchange must consist of zero or more words delimited by dots. Each word may contain the letters `[A-Z]` and `[a-z]`, and the digits `[0-9]`. The binding key follows the same rules as the routing key with the addition of `*` that matches a single word, and `#` that matches zero or more words. Thus, the binding key `*.player1.#` matches the routing keys `gameserver.player1` and `gameserver.player1.joinAction` but not `player1.gameserver`. This exchange type is stated to be optional in the AMQP specification, and is available for instance in the RabbitMQ implementation.

4.2.3 AMQP commands without any confirmations

AMQP can dispense with confirmations because it adopts an assertion model for all actions. Either they succeed, or entities have an exception that closes the channel or connection.

There are no confirmations in AMQP. Success is silent, and failure is noisy. When applications need explicit tracking of success and failure, they should use transactions.

In the TOTEM project, we do not use transactions.

4.2.4 AMQP terminology

The definitions are extracted from the specification [AMQP Consortium, 2008]:

- *Connection*: A network connection, *e.g.* in our case, a TCP/IP socket connection.
- *Channel*: A bi-directional stream of communications between two AMQP peers. Channels are multiplexed so that a single network connection can carry multiple channels.
- *Client* and *server*: The initiator of an AMQP connection or channel. AMQP is not symmetrical. Clients produce and consume messages while servers queue and route messages. The server is the process that accepts client connections and implements the AMQP message queuing and routing functions. In this document, we prefer using the terms *publishers* and *consumers* for distinguishing to two roles of clients, and the term *broker* instead of server for not confusing with system architecture concerns.
- *Content header* or *header*: A specific type of frame that describes a content's properties.
- *Content body* or *body*: A specific type of connection data that contains raw application data. Content body frames are entirely opaque; the server does not examine or modify these in any way.
- *Exchange*: The entity within the server which receives messages from producers and optionally routes these to message queues within the server.
- *Exchange type*: The algorithm and implementation of a particular model of exchange. In contrast to the "exchange instance", which is the entity that receives and routes messages within the server.
- *Message queue*: A named entity that holds messages and forwards them to consumers.
- *Binding*: An entity that creates a relationship between a message queue and an exchange.
- *Routing key*: A virtual address that an exchange may use to decide how to route a specific message.
- *Durable*: A server resource that survives a server restart.
- *Persistent*: A message that the server holds on reliable disk storage and must not lose after a restart.

- *Consumer*: A client application that requests messages from a message queue.
- *Producer*: A client application that publishes messages to an exchange.
- *Virtual host*: A collection of exchanges, message queues and associated objects. Virtual hosts are independent server domains that share a common authentication and encryption environment.
- *Subscription*: Usually a request to receive data from topics; AMQP implements subscriptions as message queues and bindings.

4.3 Presentation of RabbitMQ

RabbitMQ (`http://www.rabbitmq.com`) is an open source implementation licensed under the Mozilla Public License of the AMQP specification, version 0.9.1. It is supported by SpringSource, a division of VMware, which is an active contributor to the AMQP Consortium. For more details on the conformance to the specification, please refer to the following URL: `http://www.rabbitmq.com/specification.html` Through adapters, AMQP supports XMPP, SMTP, STOMP and HTTP for lightweight web messaging. The project is very active: versions 2.0.0, 2.1.0, 2.2.0, 2.3.0, and 2.4.0 were delivered in August 2010, September 2010, October 2010, November 2010, February 2011, and March 2011, respectively; several dozens of mails are exchanged on the mailing list per day. The core of the broker is programmed in Erlang and clients exist for many languages (Java, Ruby, Python, .NET, PHP, Perl, C/C++, Erlang, Lisp, and Haskell), many operating systems (Unix-like OSes, Windows, MacOSX, and OpenVMS), and developers make sure that RabbitMQ does work on Amazon EC2 Ubuntu AMIs. The documentation on the Web site is plentiful. The project proposes a library for Java clients. The community proposes several clients for Python; Pika is chosen for being alive (the authors are very reactive on the RabbitMQ mailing list), for being the chosen solution for implementing a tutorial on AMQP available on the RabbitMQ Web site, for being the chosen solution for the excerpts of code in the book [Videla and Williams, 2011] and for its mailing list being the mailing list of RabbitMQ.

RabbitMQ architecture allows for the extension of the broker functionalities with a plug-in mechanism: for instance, a management / monitoring API over HTTP, along with a browser-based UI; a plug-in that shovels messages from a queue on one broker to an exchange on another broker; authentication / authorization plug-ins (LDAP and SSL). For the complete list of supported and experimental plug-ins, please refer to the following URL: `http://www.rabbitmq.com/plugins.html`. In addition, RabbitMQ is complemented with non AMQP functionalities, for example clustering for better scalability.

4.4 Presentation of Node.js

Node.js (`http://www.nodejs.org`) is a JavaScript platform for easily building fast, scalable network applications. It uses an event-driven, non-blocking I/O model which fits the requirements of data-intensive real-time applications that run across distributed devices. It is licensed under the MIT License. Node.js has a native support of the WebSocket protocol, which could be a nice-to-have feature for next versions of the communication middleware. However, for the needs of Master and Spectator JavaScript applications, performances of AJAX requests are sufficient. Several points have motivated the use of Node.js to enable communication between JavaScript Web applications and the RabbitMQ broker:

- Implementing a RabbitMQ plugin to enable communication with JavaScript applications is questionable. Indeed, a defective plugin can possibly corrupt the behaviour of the broker. In addition, maintenance of a plugin is quite expensive, as a new version of the plugin has to be built for each new version of the RabbitMQ broker.

- Node.js has an AMQP library, which is quite similar to the Python one (Pika). Hence, even developers with no skills in JavaScript or in Web development will easily be able to maintain, and possibly to add new features to the library.

- Building a proxy in JavaScript, instead of using PHP or Ruby, has the advantage of using the same language on both clients and proxy sides. In TOTEM, avoiding the use of another language, in addition to Java, Python and JavaScript facilitates the understanding of the source code.

5 Architecture of the communication infrastructure

Figure 6 displays the entities that take part into the system architecture. This architecture corresponds to the following design decisions:

- The game server and the game logic servers (of the game instances) are in separate processes. The game server can be an application inserted into the TOTEM Django game server.
- Game logic servers are the "server" entities running the logic of the game instances.
- There is one RabbitMQ server that includes all the brokers of separate virtual hosts, one per game instance. RabbitMQ scales very well in terms of resources (connections, channels, exchanges, queues, and bindings). If necessary, RabbitMQ proposes a clustering extension, possibly at a (possibly much) higher cost of system administration. We have not investigated that opportunity for TOTEM.
- Brokers are in different virtual hosts for better isolation and separation of concerns.
- All the users join the game server via XML-RPC calls and are granted a queue to receive messages in the game instance broker.
- Every game master, player, and spectator opens two connections, the first one is a XML-RPC connection to access the game server for the login (as described in Figure 7) and the second one is an AMQP connection to access the game instance broker (as described in Figure 8).
- A Node.js proxy is used by JavaScript applications both for the login with XML-RPC, and for the exchanging of messages with AMQP. During the login phase, JavaScript applications use this proxy to execute their XML-RPC calls on the game server (cf. Figure 7). Once they are logged in, they use the proxy to establish an AMQP connection. Then, these Javascript applications are able to publish and to consume messages using HTTP requests (cf. Figure 8).

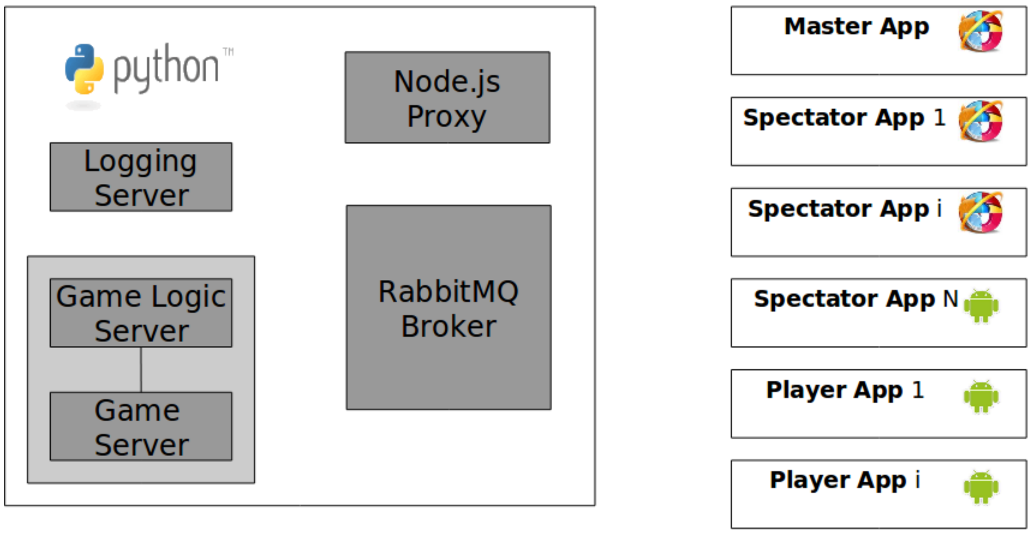

Figure 6: System architecture with a focus on the entities

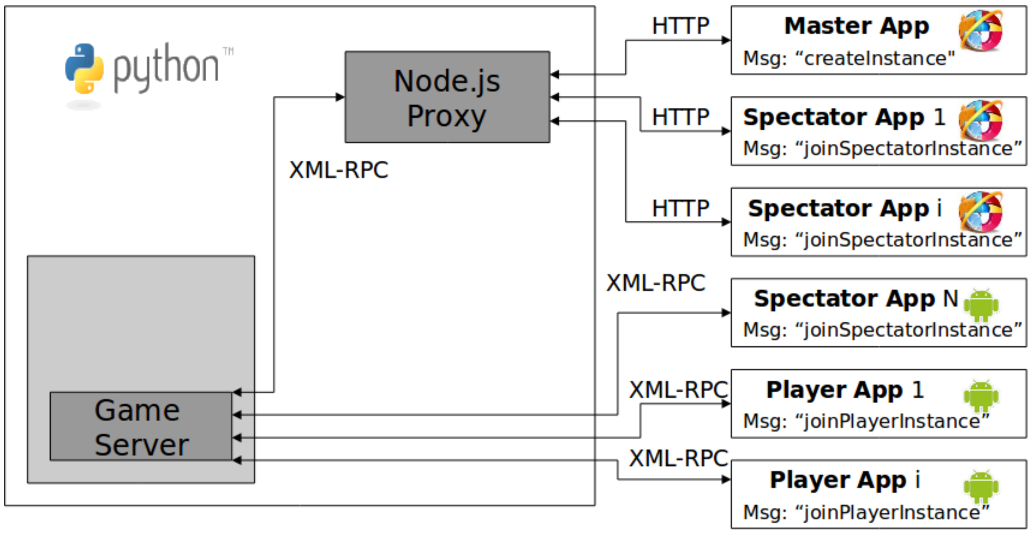

Figure 7: System architecture with a focus on the login phase using XML-RPC

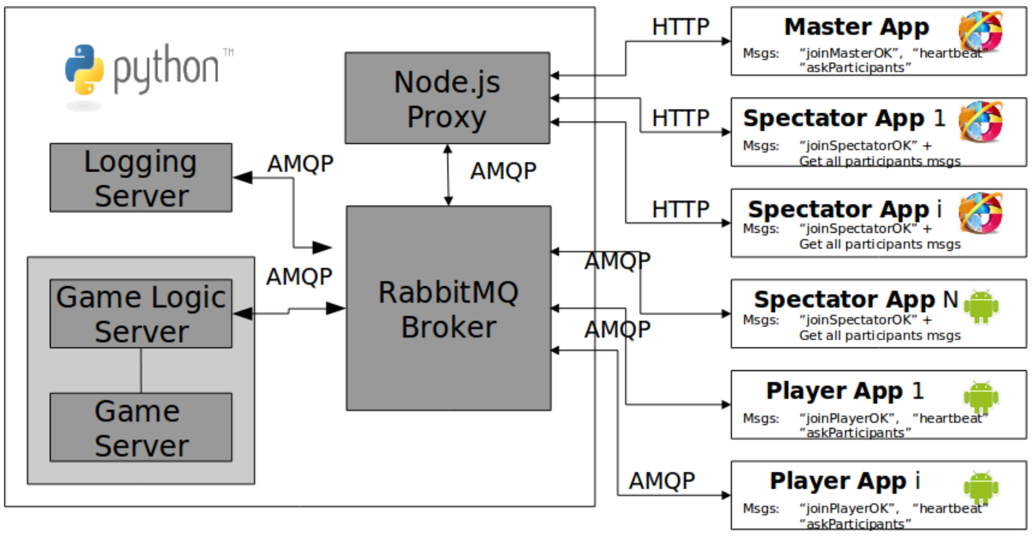
python
Node.js Proxy
Logging Server
AMQP
AMQP
RabbitMQ Broker
AMQP
Game Logic Server
Game Server
HTTP
HTTP
HTTP
AMQP
AMQP
AMQP
Master App
Msgs: "joinMasterOK", "heartbeat" "askParticipants"
Spectator App 1
Msgs: "joinSpectatorOK" + Get all participants msgs
Spectator App i
Msgs: "joinSpectatorOK" + Get all participants msgs
Spectator App N
Msgs: "joinSpectatorOK" + Get all participants msgs
Player App 1
Msgs: "joinPlayerOK", "heartbeat" "askParticipants"
Player App i
Msgs: "joinPlayerOK", "heartbeat" "askParticipants"

Figure 8: System architecture with a focus on the exchanging of messages using AMQP

6 Detailed design of the communication infrastructure

This section details the design of the communication infrastructure: the part built over the XMLRPC standard (cf. Figure 9) and the part built over the AMQP standard (cf. Figure 10).

In Figure 9, the game server receives the login calls of the end-users. The game master calls for the creation of the game instance. When created, the game logic server of the game instance receives the calls forwarded by the game server for opening the connection to the AMQP channels. At the end of the creation and login phase, the AMQP communication infrastructure is as depicted in Figure 10.

The design decisions concerning AMQP resources are the following ones:

- Every entity that communicates with the broker, either end-user applications (PlayerApplication, MasterApplication...) or processes (GameServer, LoggingServer...) matches with an AMQP user on the broker side. As a game instance is represented by a virtual host, permissions are granted to users to access the instance(s) they are logged in. Game instances are contained into distinct partitions, without any possible inter-communication. However, a user can repeat its login procedure in several game instances. It can then be able to communicate separately to several game instances.

- A topic exchange is used by the GameLogicServer in order to enable the macthing between messages and routing keys. The routing key policy used by the middleware associates each message to a sender, a recipient, an action kind and an action name. Furthermore, with the meta-characters "*" and "#", it is quite easy to enable broadcast communication, or subscription to all the users' queues.

- For each user, a queue is declared and named with the user's login name. The queue is then bound to the topic exchange GameInstance. For example, a queue named *Player1* is declared for user Player1. This queue is bound to the exchange GameInstance with the routing key `*.Player1.*.*`. It can be understood as "receive all the messages from any sender which are for Player1". Acknowledgment flags of queues are set to true: this means that the broker won't send any other message to a client as long as the previous one hasn't been acknowledged. This property is particularly useful for the Node.js proxy: if the proxy crashes, all the messages for a JavaScript client stay on the broker side.

The design and implementation decisions concerning AMQP message exchanges:

- The style of programming is by exchanging messages and reacting to messages. This is ensured by the use of a state machine. In the sequel of the document, the protocols of the state machines are sequences of messages with their type and content, and the reactions are the actions executed when receiving messages.

- The Python TOTEM library is dedicated to the applications running on the game logic server and the logging server, thus using Pika in the continuous-passing programming style (with callbacks) in order to better scale.

- The Java TOTEM library is dedicated to mobile Android applications, thus using the RabbitMQ Java Client with the blocking-calls programming style in order to limit multithreading and asynchrony.

- The JavaScript TOTEM library is dedicated to Web applications, and communicates with the RabbitMQ broker through a proxy. The proxy itself is built with the JavaScript framework Node.js, which uses an AMQP library to publish and consume messages from the broker. Messages received by the proxy and dedicated to a particular Web application are retrieved by this application via AJAX requests using long polling. To publish messages, Web applications send them to the proxy in GET requests, and the proxy publishes those messages to the broker on the AMQP connection matching the Web application.

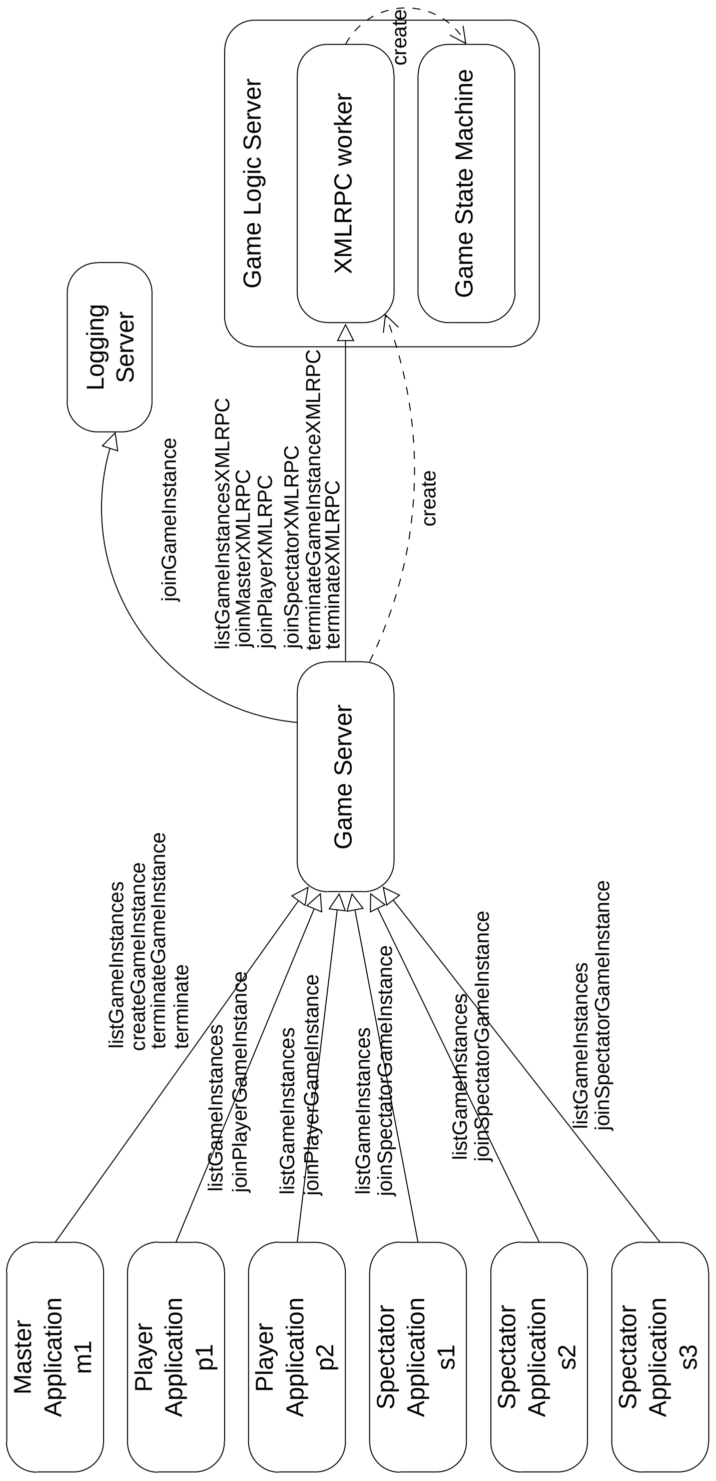

Figure 9: Architecture of the XML-RPC communication infrastructure

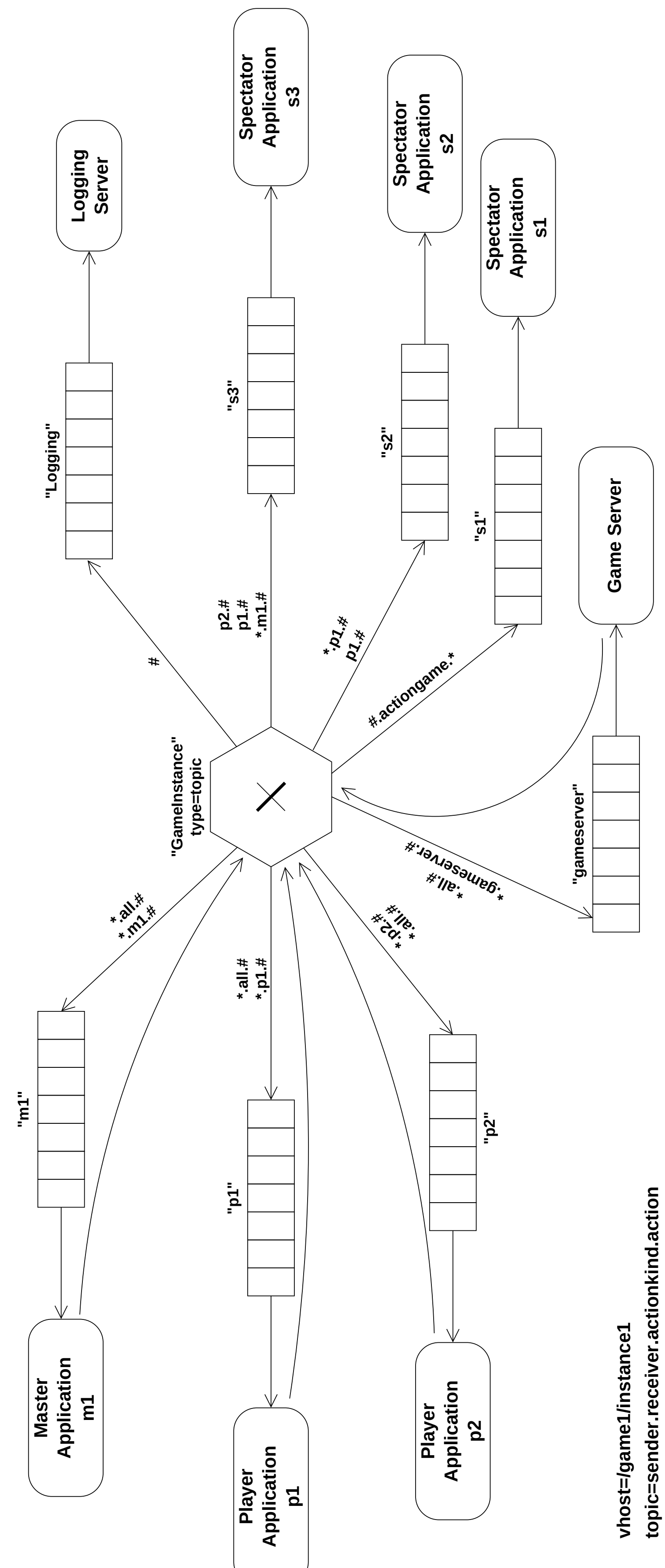

Figure 10: Architecture of the AMQP communication infrastructure

- Python, Java and JavaScript idioms are built to obtain extensible state machines (cf. Section 7).
- The GameLogicServer, which uses the Python library for AMQP Pika, cannot publish messages in separate threads, as the channel used is not thread-safe. For Java and Android clients, with the version of the AMQP client used (2.7.0 or greater), there isn't anymore such a constraint. When using JavaScript, either on a Web application or on the Node.js proxy, there isn't such a constraint neither, since JavaScript is a single-threaded language.
- As stated in the AMQP standard, when a problem happens, the connection is closed. In the Java TOTEM library, if an exception is thrown when connecting, publishing or consuming, then the connection is closed and re-opened in order to retry the operation (up to a maximum number of retries). For Web applications, a maximum number of retries is also given, but it is only used for the consumption of messages. If a disconnection between the application and the proxy occurs, the Web application tries to reconnect to the proxy to receive messages until the maximum number of retries is reached. For the publication of messages, it's up to the Web browser to ensure the sending of the GET request as soon as the network is reachable again.

The limitations of the current design and implementation, which correspond to the example application described in Section 10, are the following ones:

- No tests have been performed with several game logic servers. In the example application described in Section 10, there is one virtual host for the game logic server of the sole game instance.
- There is no defense programming such as checking whether the game instance is already created / started, the user (game master, player, or spectator) has already joined the game instance, a spectator is already a player. These kinds of verifications are business application specific and should for instance lead to the definition of many new message types and exception classes.

7 API of the communication infrastructure

The API of the communication infrastructure using RabbitMQ is provided in Java for Android and J2SE, in JavaScript for Web applications, and in Python for the the GameServer, the GameLogicServer and the LoggingServer. Section 7.1 presents the API for Android clients, Section presents the API for JavaScript clients, and Section 7.3 presents the API for Python clients.

7.1 Android Java idioms for player, game master and spectator applications

The idioms are the same for player, game master and spectator applications. But depending on client application, some class names vary. For example, the class GameMasterState is called PlayerState for the player application and SpectatorState for the spectator application. In this section, we provide the idioms for the PlayerMasterAndroid.

7.1.1 Class `PlayerApplication`

The class `PlayerApplication` is the `Activity` of the Android application. It is used to:

1. Load configuration data from the file `xmlrpc.properties`:

   ```
   gameServerXMLRPCHost        localhost
   gameServerXMLRPCPort        8888
   [...]
   ```

2. Load configuration data from the file `rabbit.properties`:

   ```
   gameServerUserName          gameserver
   gameLogicServerUserName     gamelogicserver
   loggingServerUserName       loggingserver

   gameLogicServerBrokerHost   localhost
   gameLogicServerBrokerPort   5672
   gameLogicServerExchangeType topic
   gameLogicServerExchangeName GameInstance
   [...]
   ```

3. Create and set the logger, used to display log messages on the device screen:

   ```
   Util.setLogger(this);
   ```

4. Execute the `PlayerTask`

   ```
   playerTask = new PlayerTask(PlayerApplication.this);
   playerTask.execute();
   ```

7.1.2 Class `PlayerTask`

The class `PlayerTask` allows to execute the XML-RPC call for the login and to compute the response of the GameServer. Note that we have decided to make this class extend the class AsyncTask of the Android SDK to enable the displaying of information messages during the XML-RPC call. However, if you consider that such messages are not required by your application, you can organize your code without a class extending AsyncTask. The class `PlayerTask` consists in the following steps:

1. Log-in to the game server, and either create a game instance by an XML-RPC call of the method `createGameInstance` or join an existing game instance by an XML-RPC call of the method `joinGameInstance`.

2. Create the game logic state managed by the client application:

```
state = new PlayerState();
```

3. Instantiate the ChannelManager to communicate with the broker and assign the game logic state and the lists of actions of the state machine of the client:

```
state.channel = ChannelsManager.getInstance(state,
                                            MyListOfGameLogicActions.ListOfActionsMaps);
```

4. Publish a message for announcing the joining to the broker, the second argument being the content of the message (*e.g.*, `"Publish the String "lisa,mygamename,myinstancename""`):

```
state.channel.publishToGameLogicServer(state,
                                       JoinAction.JOIN_PLAYER,
                                       state.login
                                       + ....getProperty("bodySeparator")
                                       + state.gameName
                                       + ....getProperty("bodySeparator")
                                       + state.gameInstanceName);
```

7.1.3 State machine enumeration type for the game logic of the player

The state machine for the game logic is specified into two sorts of enumeration types: the first sort enumeration type specifies the lists of action kinds (cf. Listing 1) and the second enumeration type specifies the actions per action kinds (cf. Listing 2 and Listing 3). The protocol of the game logic specified in the TOTEM library is complemented by the list of actions specified in these classes.

Listing 1: List of action kinds of the state machine for the game logic of the player

```
public enum MyListOfGameLogicActions {
  MY_FIRST_ACTION_KIND(MyFirstActionKind.actionMap),
  MY_SECOND_ACTION_KIND(MySecondActionKind.actionMap);

        // Ignore the code below. Just make sure it is present in all your enums.
        // The copy and paste is due to a limitation of Java enums (no inheritance).
```

7.1.4 Game logic protocol of the state machine

In Listing 4, the protocol of the state machine for the game logic is specified as a set of static methods with three arguments: the state of the client, the header of the message received and the content of the message. The header of the message is the routing key of the message: four strings indicating the emitter, the addressee, the action kind and the action.

7.1.5 Game logic state

The game logic state is defined in the TOTEM library and can be extended by client applications. All the attributes presented in Listing 5 are public.

Listing 2: Player's actions of the first action kind

```
public enum MyFirstActionKind implements PlayerActionInterface {
  MY_FIRST_ACTION("myFirstAction") {
    public Object execute(PlayerState state, String[] header, String body)
        throws ActionInvocationException {
      return MyGameLogicProtocol.myFirstAction(state, header,body);
    }
  },
  MY_SECOND_ACTION("mySecondAction") {
    public Object execute(PlayerState state, String[] header, String body)
        throws ActionInvocationException {
      return MyGameLogicProtocol.mySecondAction(state, header, body);
    }
  },
  MY_THIRD_ACTION("myThirdAction") {
    public Object execute(PlayerState state, String[] header, String body)
        throws ActionInvocationException {
      return null;
    }
  };
  public final static int KIND_NUMBER = 100;
  public final static int LOWER_ACTION_NUMBER = 0;
  public final static int UPPER_ACTION_NUMBER = 1000;
  // Ignore the code below. Just make sure it is present in all your enums.
  // The copy and paste is due to a limitation of Java enums (no inheritance).
  [...]
}
```

Listing 3: Player's actions of the second action kind

```
public enum MySecondApplicationInstanceActionKind implements PlayerActionInterface {
  MY_FOURTH_ACTION("myFourthAction") {
    public Object execute(PlayerState state, String[] header, String body)
        throws ActionInvocationException {
      return MyGameLogicProtocol.myFourthAction(state, header, body);
    }
  },
  MY_FIFTH_ACTION("myFitfhAction") {
    public Object execute(PlayerState state, String[] header, String body)
        throws ActionInvocationException {
      return null;
    }
  };
  public final static int KIND_NUMBER = 101;
  public final static int LOWER_ACTION_NUMBER = 0;
  public final static int UPPER_ACTION_NUMBER = 1000;
  // Ignore the code below. Just make sure it is present in all your enums.
  // The copy and paste is due to a limitation of Java enums (no inheritance).
  [...]
}
```

Listing 4: Game logic's protocol of the state machine for the game logic

```
public class MyGameLogicProtocol {
  public static Object myFirstAction(PlayerState state, String[] header, String body) {
    Util.println("Player" + state.login + "]" + body);
    return null;
  }
  public static Object mySecondAction(PlayerState state, String[] header, String body) {
    Util.println("[Player" + state.login + "]" + body);
    return null;
  }
  public static Object myFourthAction(PlayerState state, String[] header, String body) {
    Util.println("[Player" + state.login + "]" + body);
    return null;
  }
}
```

Listing 5: Player's state of the state machine

```
public class PlayerState {
    public String     login;
    public String     password;
    public String     gameName;
    public String     gameInstanceName;
    public String     virtualHost;
    public String     exchangeName;
    public ChannelsManager channelsManager;
    public int        numberOfRetries;
    public boolean    hasConnectionExited;
    connectionExit();
}
```

7.2 JavaScript idioms for game master and spectator applications

The idioms are the same for game master and spectator applications. Here, we just provide the idioms for the game master application.

7.2.1 `master.js` file

The master.js file represents the game master application, and uses the TOTEM API in JavaScript. It consists in the following steps:

1. Loggin to the game server, and creation of a game instance by an XMLRPC to the method `createGameInstance`.
2. Creation the game logic state managed by the client application:

   ```
   gameMasterState = new State(‘‘login’’,’’password’’,’’gameName’’,’’instanceName’’);
   ```

3. Implementation of the `onExitConnection()` function triggered by the API when the connection exits:

   ```
   function onExitConnection(){
     // update the display to show the end screen
     showEnd();
   }
   ```

4. Publish the message for announcing the joigning to the game instance broker, the second argument being the content body of the message (*e.g.*, `lisa,mygamename,myinstancename`):

   ```
   publishToGameLogicServer(gameMasterState, "join.joinMaster", "lisa,Tidy-City,Instance-1");
   ```

7.2.2 State machine actions for the game logic

The state machine for the game logic is specified into actions kinds and actions. Actions kinds can be considered as name spaces to facilitate usage of actions. Listing 6 gives an example. Please refer to section 9.1.4 for further details.

Such an action kind have to be declared to the game logic state:

```
gameMasterState.listOfActions.myFirstActionKind = new MyFirstActionKind();
```

7.2.3 Game logic's State class

The game logic's State class is defined in the TOTEM library. All the attributes presented in Listing 7 are public.

Listing 6: GameMaster's actions of the first action kinds

```
function MyFirstActionKind () {}

MyFirstActionKind.prototype.myFirstAction = function (state, publisher, consumer, message){
        println("My first action","Message received from "+publisher+": "+message);
};
```

Listing 7: GameMaster's state

```
function State(login, password, gameName, instanceName, heartbeat, maxRetry){
    this.login = login;
    this.password = password;
    this.gameName = gameName;
    this.instanceName = instanceName;
    if (typeof heartbeat == "number"){
        this.heartbeat = heartbeat;
    }else{
        this.heartbeat = DEFAULT_HEARTBEAT_TASK_PERIOD;
    }
    if (typeof maxRetry == "number"){
        this.maxRetry = maxRetry;
    }else{
        this.maxRetry = DEFAULT_MAX_RETRY;
    }
    this.listOfActions = {  join: new JoinAction(),
                            presence: new PresenceAction(),
                            lifecycle: new LifeCycleAction()};
    [...]
}
```

7.3 Python idioms for the game logic server

This section details only the idioms for the game logic server. The protocol of the game logic server specified in the TOTEM library is complemented by the list of actions specified in the listings 8, 9, and 10.

Listing 8: List of action kinds of the state machine for the game logic server

```
MyListOfActions = ListOfActionsEnumeration("MyListOfActions",
    [("myFirstActionKind", MyFirstActionKind),
     ("mySecondActionKind", MySecondActionKind)
     ])
```

7.3.1 Game logic server state

The game logic server's state is defined in the TOTEM library and can be extended by client applications. All the attributes presented in Listing 11 are public.

7.3.2 Game logic server's protocol of the state machine

In Listing 12, the protocol of the state machine for the game logic server is specified as methods with three arguments: the state of the game logic server, the header of the message received and the content of the message.

Listing 9: Game logic server's actions of the first action kinds

```
def doNothing(state, header, body):
    pass

MyFirstActionKind = GameLogicActionEnumeration("myFirstActionKind", 100, 0, 1000,
    [("myFirstAction", myFirstAction),
     ("mySecondAction", mySecondAction),
     ("myThirdAction",doNothing)
     ])
```

Listing 10: Game logic server's actions of the second action kinds

```
def doNothing(state, header, body):
    pass

MySecondActionKind = GameLogicActionEnumeration("mySecondActionKind", 101, 0, 1000,
    [("myFourthAction", myFourthAction),
     ("myFitfhAction", myFitfhAction)
     ])
```

Listing 11: Game logic server's state

```
class MyState(GameLogicState):
    mynewstatevariable = None
class GameLogicState(object):
    createGameInstanceSemaphore = None
    vhost = None
    connection = None
    channel = None
    gamelogiclistofactions = None
    gameName = None
    instanceName = None
```

Listing 12: Game logic server's protocol of the state machine

```
def myFirstAction(state, header, body):
    print '␣[GameLogicServer]␣React␣to␣myFirstAction␣message'

def mySecondAction(state, header, body):
    print '␣[GameLogicServer]␣React␣to␣mySecondAction␣message'

def myFourthAction(state, header, body):
    print '␣[GameLogicServer]␣React␣to␣myFourthAction␣message'

def myFitfhAction(state, header, body):
    print '␣[GameLogiceServer]␣React␣to␣myFitfhAction␣message'
```

8 Organisation of the source code

The source code provided into the directory structure TOTEM.CommunicationMiddleware/Sources is organised as follows. Every directory contains a file `readme.txt`.

```
Sources
|-- IntegrationExampleApplication      # application presented in Section 9
|   |-- GameLogicServer                # game logic of the game
|   |-- GameServer                     # Python server part managing the broker
|   |-- MasterApplication              # J2SE game master application
|   |-- MasterApplicationJavascript    # Javascript game master application
|   |-- NodeJsProxy                    # Node.js proxy used by Javascript applications
|   |-- PlayerApplication              # J2SE player application
|   |-- PlayerMasterAndroid            # Java Android application acting both as player and master
|   |-- SpectatorApplication           # J2SE spectator application
|   |-- SpectatorApplicationJavascript # Javascript spectator application
|   `-- TerminationApplication         # Python application to call for termination
|-- RabbitMQTOTEMLibrary               # Code extracted to be in the TOTEM library
|   |-- Java
|   |   |-- common
|   |   |-- gamemasterapplication
|   |   |-- playerapplication
|   |   `-- spectatorapplication
|   |-- JavaScript
|   |    `-- external
|   `-- Python
|       `-- eu
|           `-- telecomsudparis
|               `-- rabbitmq
|                   |-- applicationinstance
|                   `-- configuration
`-- TutorialExamples       # RabbitMQ tutorial, cf. http://www.rabbitmq.com/getstarted.html
    |-- Android
    |   |-- Step1, |-- Step2, |-- Step3, |-- Step4, |-- Step5, `-- Step6
    |-- J2SE
    |   |-- Step1, |-- Step2, |-- Step3, |-- Step4, |-- Step5, `-- Step6
    |-- Javascript
    |   |-- Step1, |-- Step2, |-- Step3, |-- Step4, |-- Step5, `-- Step6
    `-- Pika
        |-- Step1, |-- Step2, |-- Step3, |-- Step4, |-- Step5, `-- Step6
```

9 Tutorial: understanding the concepts of the middleware to publish and consume messages

This section is a tutorial for explaining the concepts of the middleware and of its client applications (located in the directory `TOTEM.CommunicationMiddleware/Sources/IntegrationExampleApplication` of the TOTEM SVN), and the use of those concepts to publish and consume GPS coordinates using the AMQP part of the TOTEM Communication Middleware. It is also the opportunity to briefly presents the different XML-RPC methods used to interact with the game server.

9.1 In a nutshell

9.1.1 Game server and Game Logic server

A client of the middleware can communicate with two different entities: the Game server and the Game Logic server. The Game server represents the lobby (term used in the jargon of multiple-players video games): it allows the listing, the creation, the joining and the termination of game instances. The Game Logic server represents a single game instance. Two different protocols should be used depending on the entity to communicate with: communication with the Game server is ensured by XML-RPC calls, whereas communication with the Game Logic server is based on AMQP.

9.1.2 Interacting with the game server using XML-RPC

A client should first either create a game instance, or list the already existing game instances and join one. To perform the XML-RPC calls, a Java/Android application calls the Game server using XML-RPC. For JavaScript applications, the methods are provided by the API: they allow to send a request to the Node.js proxy which executes the XML-RPC calls to the GameServer. Here follows the methods that can be called on the Game server:

- `Object[] listGameInstances(gameName)`

  to retrieve the list of existing instances for a given game (Object[] for Java clients, whose elements can be cast in String - tuple for Android and JavaScript clients);

- `Boolean createGameInstance(login, password, gameName,instanceName)`

  to create a game instance for a given game, and to join it;

- `Boolean joinPlayerGameInstance(login, password, gameName, instanceName)`

  to make a player application (Java or Android) join an existing game instance;

- `Boolean joinSpectatorGameInstance(login, password,gameName,instanceName, observationKey)`

  to make a spectator application (Java or JavaScript) join an existing game instance ,observationKey is used to filter messages to consume;

- `Boolean terminateGameInstance(gameName, instanceName)`

  to terminate a given instance of a given game;

- `Boolean terminate()`

  to terminate the game server, and hence every existing game instances. Note: clients don't need special permissions to call this method.

9.1.3 Publishing messages

Once a client has joined a game instance, the publication and the consumption of messages within the game instance is ensured using the AMQP protocol. The TOTEM Communication Middleware provides high-level methods hiding the complexity of AMQP.

Messages can be published to a given user (designed by its login) or to the Game Logic server, or can be broadcast to all the users of a game instance and to the Game Logic server. The three matching (publish, publishToGameLogicServer, publishToAll) publication methods use the following parameters:

- the **state** of the state machine of the user, which contains the information required for the publication of a message (login, game name, instance name...),
- a String representing the **content** of the message,
- an **action kind** and an **action**. Their role is described in Section 9.1.4.

As it is already mentioned in Section 6, the **Game Logic server programmed in Python should not publish messages in separate threads**, since the AMQP mechanism for the publication of messages is not thread-safe. There isn't such limitation on Java Android and JavaScript applications.

9.1.4 Computing messages: actions and action kinds

Since clients of the TOTEM Communication Middleware don't have to handle the low-level details of reception of messages, they just need to define a behaviour on the consumption of these messages. This is why actions and action kinds are made for. An action represents a particular behaviour, whereas an action kind defines a name space for a set of actions of a whole protocol. When a client publish a message to another client, it must give both the action kind and the action matching with the message. When the other client consumes the message, if the action kind and the action are defined on its state machine, the corresponding method will automatically be triggered by the TOTEM Communication Middleware. Otherwise, the messages are printed nevertheless. The implementation of actions and action kinds are different for Java Android applications, JavaScript applications and for the Game Logic server in Python (please refer to Chapter 7 for further details). Let's see the example of a JavaScript application which has defined the following action kind and action:

```
function LocationActionKind () {}

LocationActionKind.prototype.sendCoordinates = function (state, publisher, consumer, message){
      println("Send coordinates action","New coordinates from "+publisher+": "+message);
};
```

and which has stored the action kind on its state machine:

```
myState.listOfActions.LocationActionKind = new LocationActionKind();
```

From now, every client that publishes a message to this JavaScript application, setting the action parameter of the publish method to `"LocationActionKind.sendCoordinates"` (actionkind.action) will trigger the execution of the `sendCoordinates(state, publisher, consumer, message)` function on the JavaScript application side.

9.2 Publishing and consuming GPS coordinates

In Section 9.2.1, a player periodically publish a message to another player. On the consumption of the message, the other player prints this message. In Section 9.2.2, the player broadcast its coordinates to the others players and to the game logic server. In Section 9.2.3, the master application written in JavaScript consumes the messages.

9.2.1 Point to point communication between Android devices

In the Android project `PlayerMasterAndroid`:

1. Create new actionKind and action in a Java state machine:
   - Copy the class `MyFirstActionKind` located in the package `eu.telecomsudparis.integration.player.android`, paste it in the same package, and rename it `LocationActionKind`.
     Note: If you are using an IDE like Eclipse, every references to `MyFirstActionKind` should be automatically replaced by reference to `LocationActionKind`. Otherwise, you need to replace those references by yourself.
   - In this new class:
     - The value of the attribute `nameKind` is set to `"myFirstActionKind"`. Replace this value by `"locationActionKind"`.
       * The action kind part of location messages is a string, for instance `player1.gamelogicserver.locationActionKind` followed by a string specifying the action.
     - The value of the attribute `KIND_NUMBER` is to `100`. Replace this value by `102`.
     - The name of the first enum of the list is set to `MY_FIRST_ACTION("myFirstAction")`, Replace this name with `SEND_GPS_COORDINATES("sendGPSCoordinates")`.
       * The action part of location messages is specified by that string, for instance `player1.gamelogicserver.locationActionKind.sendGPSCoordinates`.
   - The enumerations `MY_SECOND_ACTION("mySecondAction")` and `MY_THIRD_ACTION("myThirdAction")` can be removed. Note: Beware to keep a semicolon after the last enumeration, otherwise the source code does not compile.

2. Implement the required behaviour on the reception of the action:
   - In the class `MyGameLogicProtocol`, add the following static method:

     ```
     public static Object computeGPSCoordinates(String player, String coordinates) {
           Util.println(player+" GPS coordinates received : "+coordinates);
           return null;
     }
     ```

   - In the class `LocationActionKind`, complete the method `execute` of `SEND_GPS_COORDINATES`:

     ```
     public Object execute(PlayerState state, String[] header, String body)
                         throws ActionInvocationException {
         return MyGameLogicProtocol.computeGPSCoordinates(header[0],body);
     }
     ```

     Notice that `header[0]` refers to the sender of the message.

3. Register the new action kind in the list of actions:
   - Add a new enumeration in the class `MyListOfGameLogicActions`, after MY_FIRST_ACTION_KIND(MyFirstActionKind.actionMap) and MY_SECOND_ACTION_KIND(MySecondActionKind.actionMap):
     - `LOCATION_ACTION_KIND(LocationActionKind.actionMap);`

4. Simulate the publication of GPS coordinates:

Listing 13: Method to send simulated GPS coordinates

```
// simulates the sending of GPS coordinates
private void startSendGPSCoordinatesThread() {
     new Thread(){
     double latitude  = −179.8642632;
     double longitude = −179.2570048;
     // the recipient is the other player
     String recipient = state.login.equals(PlayerApplication.INSTANCE_CREATOR_NAME) ?
                                          PlayerApplication.INSTANCE_JOINER_NAME :
                                          PlayerApplication.INSTANCE_CREATOR_NAME;
     public void run() {
         while (!state.hasConnectionExited()){
             try {
                 state.channelsManager.publish(recipient,
                                              state,
                                              LocationActionKind.SEND_GPS_COORDINATES,
                                              latitude+"/"+longitude);
                 // increment the coordinates
                 latitude ++;
                 longitude ++;
             } catch (IOException e) {
                 e.printStackTrace();
             }
             // wait for 5 seconds
             try {
                Thread.sleep(5000);
             } catch (InterruptedException e) {
                Util.println("[Player" + state.login +
                                 "]␣Thread␣sleep␣was␣interrupted");
             }
         }
     }
  }.start();
}
```

- In the class `PlayerTask`, add the method presented in Listing 13:
- Complete the method `doInBackground(Void... params)` to make the client which creates the instance calling this method:

```
@Override
protected Integer doInBackground(Void... params) {
    [...]
    if(res){
        if(state.login.equals(PlayerApplication.INSTANCE_CREATOR_NAME)){
             startSendGPSCoordinatesThread();
        }
        return RESULT_OK;
    }else{
        return RESULT_ERROR;
    }
}
```

5. Test the communication between two Android devices (or emulators):

   (a) Configure and start the Game server:

   - Set the IP addresses of the GameServer and of the TerminationApplication. Replace the value of `gameLogicServerBrokerHost` in the file `GameServer/rabbitmq.properties`, and the values of `gameServerXMLRPCHost` in the files `GameServer/xmlrpc.properties` and `TerminationApplication/xmlrpc.properties`.
   - Start the server side launching the following command in a shell:

```
$ ./run_with_android_phones.sh
```

(b) Configure and run the PlayerMasterAndroid on two Android devices:
   - Set the IP addresses of the PlayerMasterAndroid. Replace the value of `gameLogicServerBrokerHost` in the file `PlayerMasterAndroid/res/raw/rabbitmq.properties` and the value of `gameServerXMLRPCHost` in the file `PlayerMasterAndroid/res/raw/xmlrpc.properties`.
   - Run the application as an Android Application on two devices.

(c) Create and join the game instance to test the sending of messages:
   - On the first device, press the menu button, and click on “Create Instance”.
   - On the second device, press the menu button, and click on “Join Instance”.
   - Once the two devices are correctly logged into the game instance, you should see on the second device the reception of new GPS coordinates from the first device every five seconds, and the displaying of the following message:

     ```
     PLAYER_1 GPS coordinates received: -167.8642632/-167.2570048
     ```

     This message means that the action `sendGPSCoordinates` of the `LocationActionKind` has been properly triggered.

(d) Terminate the instance, the Game server and the two Android applications:
   - Execute the following command in a shell:

     ```
     $ ./termination.sh
     ```

9.2.2 Broadcast between Android devices and the Game Logic server

This subsection presents how to implement the reception of GPS coordinates on the game logic server side.

1. Create the new `actionKind` and `action` in the Python state machine:

   - In the directory GameLogicServer, copy/paste the file `myfirstactionkind.py` into the new file `locationactionkind.py`
   - In this new file:
     – Rename the variable `MyFirstActionKind` into `LocationActionKind`.
     – The value of the first argument of the constructor is set to `"myFirstActionKind"`. Replace this value by `"locationActionKind"`.
     – The value of the second argument of the constructor is set to `100`. Replace this value by `102`.

2. Implement the required behaviour on the reception of the `action`:

   - In the file `myprotocol.py`, add the following method:

     ```
     def computeGPSCoordinates(state, header, body):
         print ' [GameLogicServer] GPS coordinates received from %r: %r' % (header[0], body)
     ```

   - In the file `locationactionkind.py`, the first element of the dictionary is set to `("myFirstAction", myFirstAction)`. Replace it with `("sendGPSCoordinates",computeGPSCoordinates)` to indicate that it’s the method `computeGPSCoordinates` that should be called on the reception of a message `sendGPSCoordinates`.
   - You can remove the two other elements `("mySecondAction", mySecondAction)` and `("myThirdAction",doNothing)` from the dictionnary. In this case, don’t forget to remove the comma after the first element, otherwise the source code contains a syntax error.

- Add the following import at the beginning of the file:
  `from myprotocol import computeGPSCoordinates`

3. Register the new `action kind` in the list of actions:

   - In the file `mylistofactions.py`, add the following element at the end of the dictionary:
     `("locationActionKind", LocationActionKind)`
   - Add the corresponding import at the beginning of this file:
     `from locationactionkind import LocationActionKind`

4. Make the Player application broadcast the sending of its GPS coordinates:

   - In the class PlayerTask of the PlayerApplicationAndroid, inside the `startSendGPSCoordinatesThread()` method, replace the method `publish` by the following one:

     ```
     state.channelsManager.publishToAll(state,
                                        LocationActionKind.SEND_GPS_COORDINATES,
                                        latitude+"/"+longitude);
     ```

5. Test the broadcast between Android devices and the Game server:

   - Repeat the procedure described section 9.2.1 point 5.
   - On the Game server shell, you should see the reception of new GPS coordinates from the first device every five seconds, and the displaying of the following message:

     ```
     [GameLogicServer] New GPS coordinates received from 'PLAYER_1':
     '-148.8642632/-148.2570048'
     ```

     This message means that the action `sendGPSCoordinates` of the `LocationActionKind` has been properly triggered on the Game server side.

9.2.3 Adding a Master application in JavaScript

Another feature of the middleware is the possibility to add JavaScript applications acting like master or spectator applications. This Section explains how the action `LocationActionKind/ sendGPSCoordinates` is implemented in JavaScript.

1. Create new action kind and action in the MasterApplicationJavaScript:

   - add a new action kind and a new action at the end of the file `MasterApplicationJavaScript/my-master-actions.js`, after the declaration of `MyFirstActionKind` and `myFirstAction`:

     ```
     function LocationActionKind () {}

     LocationActionKind.prototype.sendGPSCoordinates = function (state, publisher,
                                                                  consumer, message){
             println(publisher,"New GPS coordinates received: "+message);
     };
     ```

2. Register the new `action kind` in the list of actions:

   - In the function `$(document).ready()` of the `master.js` file, add the following line after the instantiation of the `gameMasterState`:

     ```
     gameMasterState.listOfActions.locationActionKind = new LocationActionKind();
     ```

3. Test the broadcast between an Android device, the Game server and the MasterApplicationJavaScript:

   (a) Make sure that you have configured the IP addresses of the PlayerMasterAndroid and of the GameServer as described in Section 9.2.1, item 5.

   (b) Configure the NodeJsProxy used by JavaScript applications:

   - Set the IP addresses used for the NodeJsProxy, for the GameServer and for the RabbitMQ broker. Replace the values of `nodeProxyHost` and `gameServerHost` in the file `NodeJsProxy/resources/config.properties`, and the value of gameLogicBrokerHost in the file `NodeJsProxy/resources/rabbitmq.properties`.

   (c) Modify the PlayerMasterAndroid application to enable the sending of GPS coordinates:

   - Update the method `doInBackground(Void... params)` of the class `PlayerTask.java`, add the following commentary:

```
@Override
protected Integer doInBackground(Void... params) {
    [...]
    if(res){
        startSendGPSCoordinatesThread();
        return RESULT_OK;
    }else{
        return RESULT_ERROR;
    }
}
```

   (d) Start the server side by launching the following command in a shell:

```
$ ./run_with_master_and_spectator_javascript_applications.sh
```

   (e) Create a new game instance with the MasterApplicationJavascript:

   - Start the Master application located at the URL `http://NODE_PROXY_HOST:8001/Master` (where NODE_PROXY_HOST is the address of the variable `nodeProxyHost` in the file `NodeJsProxy/resources/config.properties`).

   (f) Join the game instance with the PlayerMasterAndroid to test the sending of messages:

   - Run the application as an Android Application on a device.
   - Press the menu button, and click on "Join Instance".
   - Once the device is correctly logged in to the game instance, you should see on the master application the reception of new GPS coordinates from the Android device every five seconds, and the display of the following message:

```
PLAYER_2 New GPS coordinates received: -179.8642632/-179.2570048
```

     This message means that the action `sendGPSCoordinates` of the `LocationActionKind` has been properly triggered.

   (g) Terminate the instance, the GameServer, the NodeJsProxy and the Android application:

   - Press the "Terminate instance" button of the Master application.
   - Execute the following command in a shell:

```
$ ./termination.sh
```

10 Example application demonstrating the integration of the technologies

This section presents an illustrative application that demonstrates the communication technologies of the TOTEM project. The section focuses on technologies, design patterns and idioms, not on “business” functionalities for game applications.

10.1 Analysis

10.1.1 Subsystems, use cases and scenarios

Figure 2 of Section 2 presents the use cases of the illustrative application. Figure 11 lists the functionalities of the Game Server and Logging Server. For the sake of simplicity, the use cases of the Master Application, the Player Application and Spectator Application are the ones accessed by these subsystems in Figure 11.

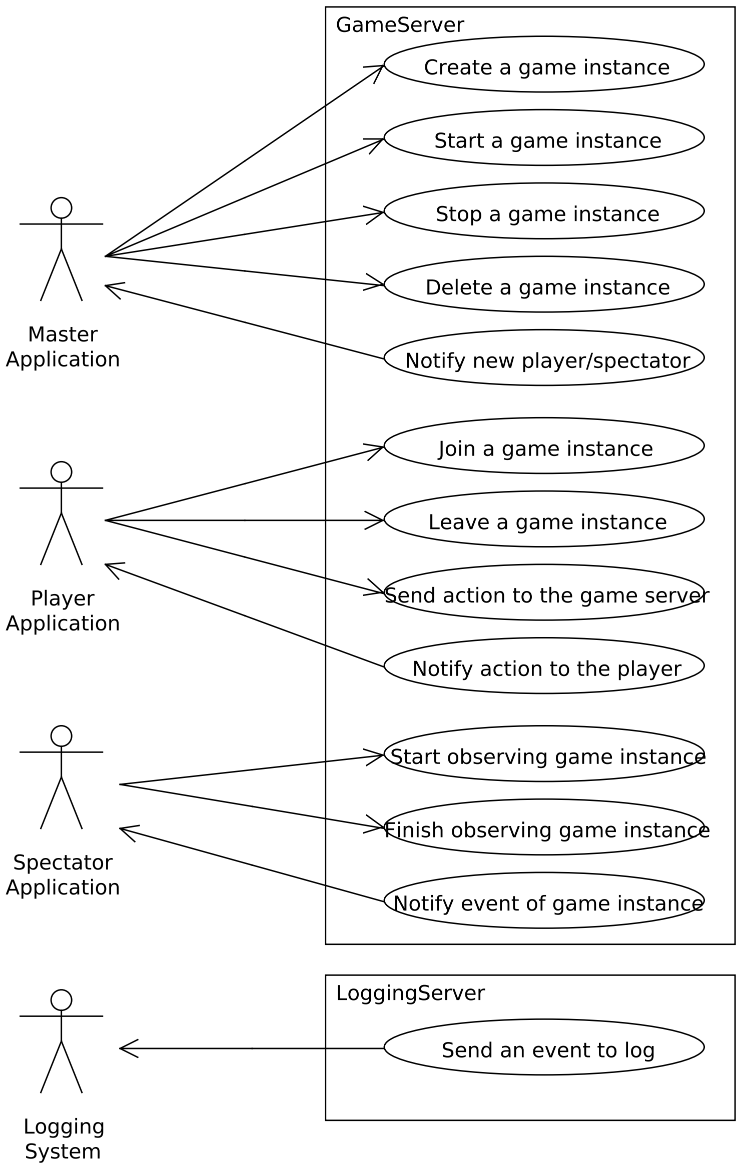

Figure 11: Use case diagram of the subsystem Game Server

The scenarios are modelled into activity diagrams:

- Figure 12 depicts the steps of the creation of the game and the game instances, and then the steps up to the beginning of the game, that is the period during which players and spectators can join the game instance. The signals are colored according to the emitter or the receiver: magenta is for game masters, cyan is for spectators and yellow is for players. Note that the game master that creates a game and the game master that creates a game instance may be two different participants; in Figure 12, they are named m_1 and m_2, respectively.
- Figure 13 explains how a player is informed about the list of game instances that she can join, and how she ask for joining the chosen game instance as player. A similar activity diagram can be drawn to model how a spectator chooses the game instance and how she ask for joining as a spectator.
- Figure 14 shows the end of a game instance when the game master asks for its closure.

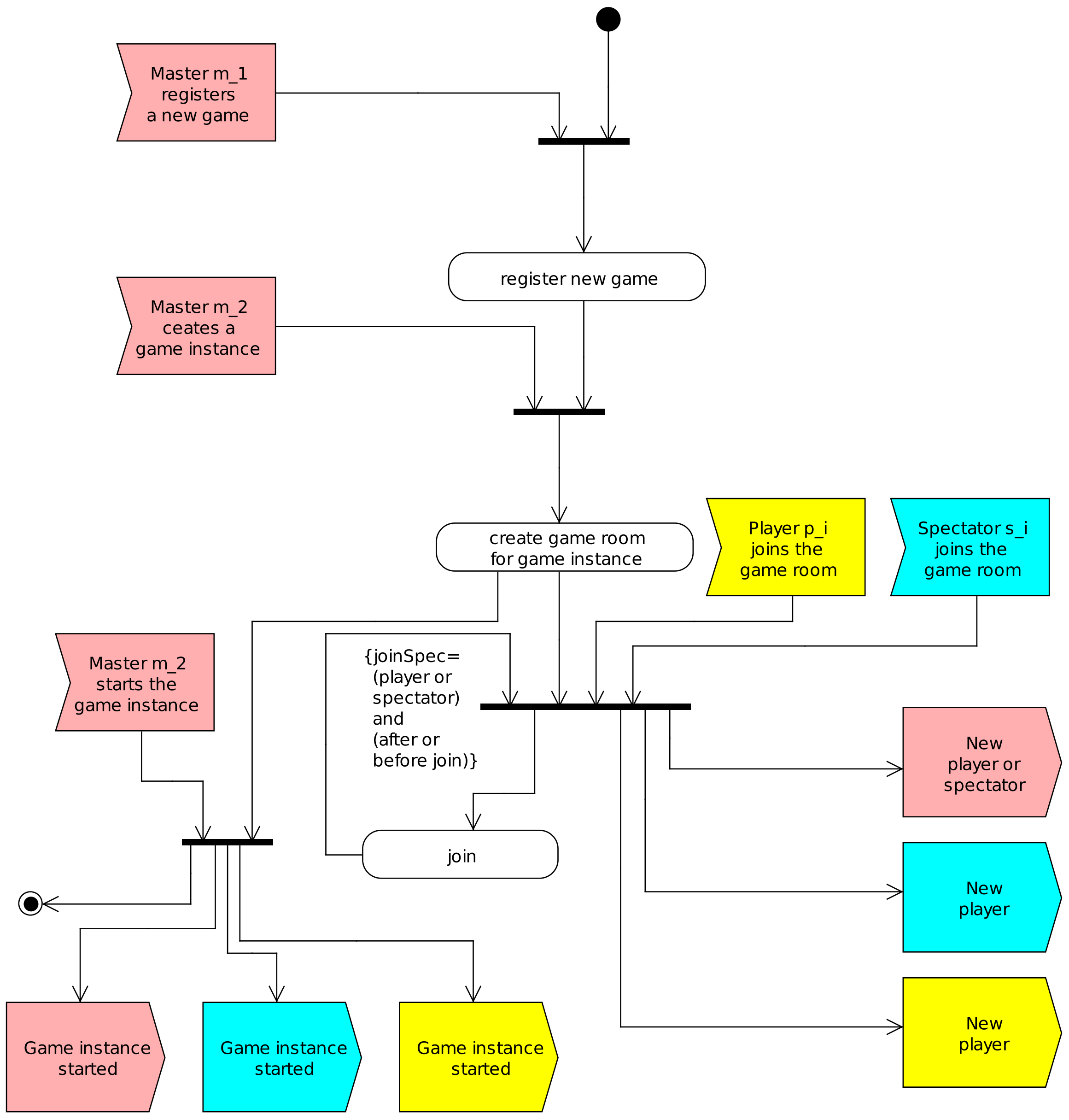
Master m_1 registers a new game
register new game
Master m_2 ceates a game instance
create game room for game instance
Player p_i joins the game room
Spectator s_i joins the game room
Master m_2 starts the game instance
{joinSpec= (player or spectator) and (after or before join)}
join
New player or spectator
New player
New player
Game instance started
Game instance started
Game instance started

Figure 12: Activity diagram of the creation of the game and the game instance, and of the joining of players and spectators

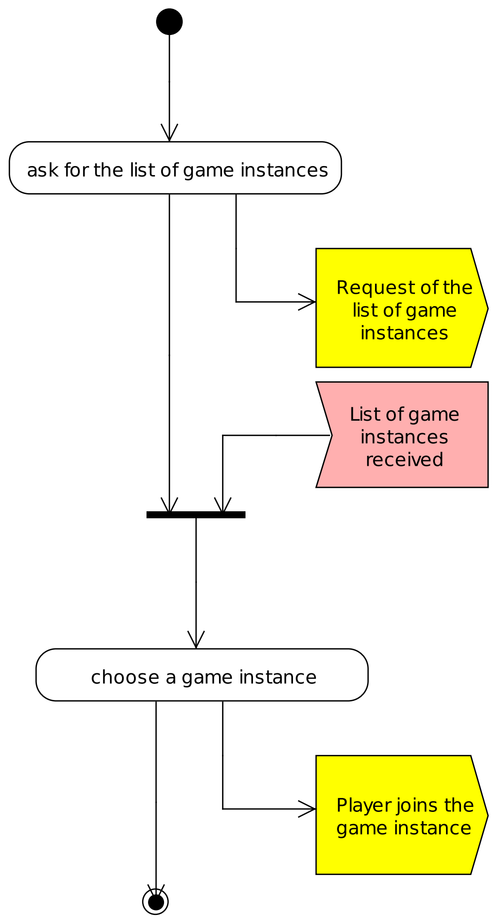
ask for the list of game instances
Request of the list of game instances
List of game instances received
choose a game instance
Player joins the game instance

Figure 13: Activity diagram of how a player is informed about and chooses a game instance

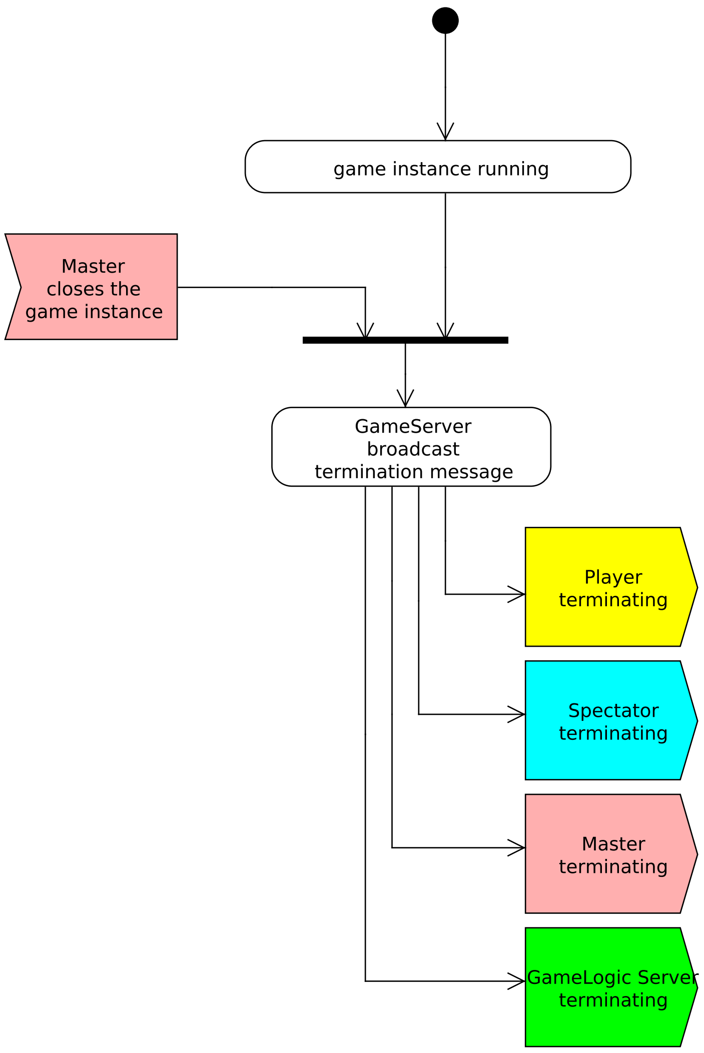
game instance running
Master closes the game instance
GameServer broadcast termination message
Player terminating
Spectator terminating
Master terminating
GameLogic Server terminating

Figure 14: Activity diagram of the end of the game instance

10.1.2 Business classes

The static aspects of the analysis of the system is presented in the class diagram of Figure 15. This model is very simple since we have simplified the business concerns in order to focus on the architectural aspects of the communication part. A Game is specified by providing a name only. It can be "played" several times (possibly in parallel), leading to several Game Instances. Each Game Instance is created and controlled by a Participant called the master, involves several Participants that are the players, and can be followed by other Participants called the spectators. A Participant can take part to several Game Instances either as master, player or spectator. A logged message, named a Log for short, is related to a Game Instance and a Participant. Not explicitly shown in this diagram, we will allow the logging of purely "technical messages" from all the subsystems and that do not concern a particular Game Instance or a specific Participant; this is why the multiplicity of the associations connected to the class Log are "$0..n$".

NB 1: In this illustrative application demonstrating the integration of the different technologies into a consistant communication architecture, we ignore rights management. In other words, every participant can be a master and every participant can be a player or a spectator of every game instance. The login and password attributes of the class Participant serves to have access to the communication infrastructure.

NB 2: In this illustrative application, for the sake of simplicity, we log only events/messages that belong to a game instance.

NB 3: In this illustrative application, no information is stored in a database, thus there is no database access. This should not be the case if the application is integrated into the TOTEM Django game server.

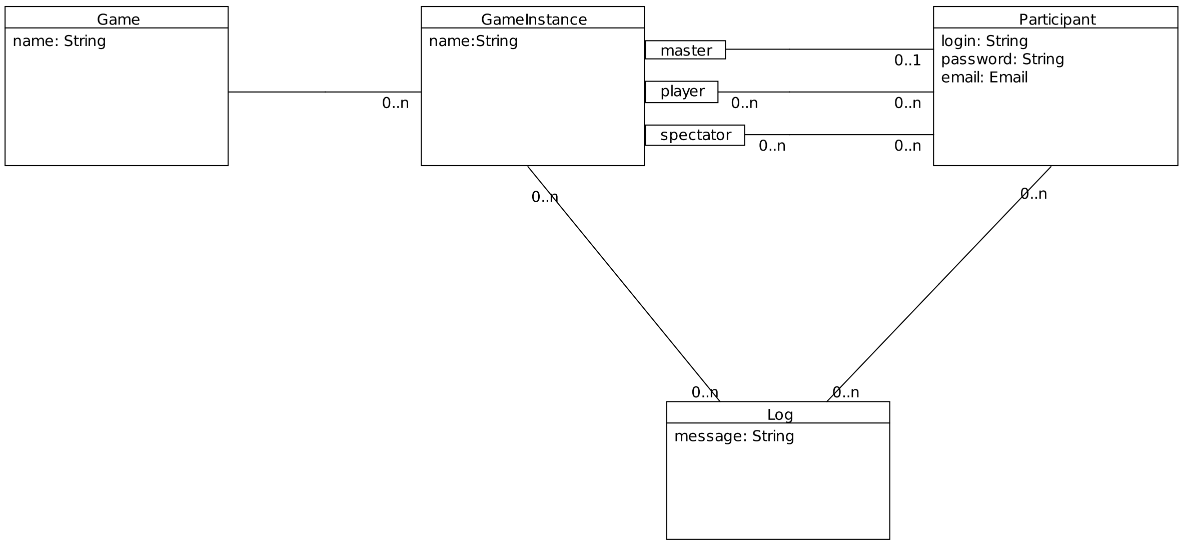

Figure 15: Class diagram of the business concepts used in the subsystems

10.2 Design of the communication architecture for the game instance

During a game instance, messages are exchanged through the event-based system whose configuration is presented in Figure 10 (a game instance with a game logic server, one game master, two players, two spectators, and the logging mechanism activated). The figure uses the following concepts from AMQP:

- The identifier of the game instance serves as the name of the virtual host. There is one virtual host per game instance so that game executions do not interfer.
- The first entity in the system is the Game Logic Server. It is responsible for creating the topic exchange and the queues.
- Every participant (game master, players, spectators, and logging services) has a dedicated queue to receive events.
- The topic exchange allows filtering with routing keys decomposed into four parts: the identifier of the sender (for instance, p1), the identifier of the receiver (for instance, m1), the kind of action transported in the event (for instance, join), and the name of the action (for instance, joinMaster).
- The Game Logic Server, the game master, and the players have two bindings: for instance `*.m1.#` for receiving messages sent only to them, and `*.all.#` for receiving broadcast messages.

The creation and the management of the game instance communication architecture is described in the following figures:

- Figure 16 presents the sequence diagram of the creation of the game instance and of the joining of the Master Application to the game instance. The Master Application sends an XMLRPC request to the Game Server. The Game Server creates the Game Logic Server process. This process is responsible for creating the AMQP communication infrastructure and creating the thread XMLRPC Worker, which is reponsible for performing the join action of the actors. The Game Server and the Game Logic Server synchronize using a semaphore so that the first join operations are not called before the virtual host and the exchange are created. Figure 19 displays the architecture after the creation of the game instance. Next, Figure 20 depicts the new architecture of the application instance when the Master Application has joined the application instance.
- Figure 17 presents the sequence diagram of the joining of a Spectator Application to the game instance. The Spectator Application sends an XML-RPC request to the Game Server that forwards it to the XMLRPC Worker thread of the Game Logic Server. Figure 21 depicts the new architecture of the application instance when the Spectator Application has joined the application instance.
- Figure 18 presents the sequence diagram of the joining a Player Application to the game. The Player Application sends an XMLRPC request to the Game Server that forwards it to the XMLRPC Worker thread of the Game Logic Server. Figure 22 depicts the new architecture of the application instance when the Player Application has joined the application instance.

NB: in all the models of this section, for the sake of simplicity, we assume that the actors know their identity and obtain the information about games and game instances from another means, for instance from the Web framework. We also assume that they have unique identities and that their login names do not contain any dot (since the login names are used for binding and routing keys).

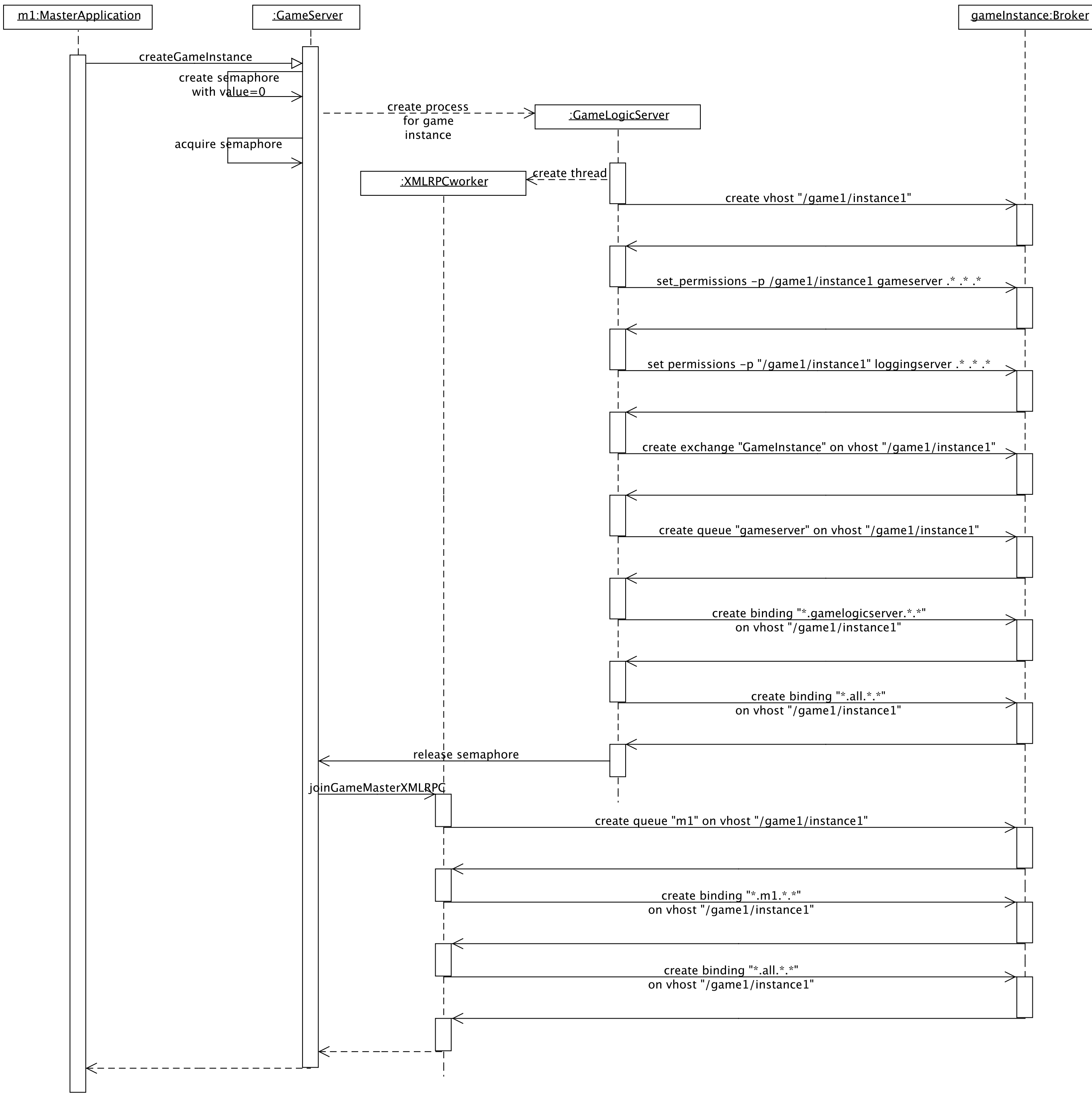
m1:MasterApplication
:GameServer
gameInstance:Broker
createGameInstance
create semaphore with value=0
create process for game instance
:GameLogicServer
acquire semaphore
:XMLRPCworker
create thread
create vhost "/game1/instance1"
set_permissions -p /game1/instance1 gameserver .* .* .*
set permissions -p "/game1/instance1" loggingserver .* .* .*
create exchange "GameInstance" on vhost "/game1/instance1"
create queue "gameserver" on vhost "/game1/instance1"
create binding "*.gamelogicserver.*.*" on vhost "/game1/instance1"
create binding "*.all.*.*" on vhost "/game1/instance1"
release semaphore
joinGameMasterXMLRPC
create queue "m1" on vhost "/game1/instance1"
create binding "*.m1.*.*" on vhost "/game1/instance1"
create binding "*.all.*.*" on vhost "/game1/instance1"

Figure 16: Sequence diagram of the creation of the game instance

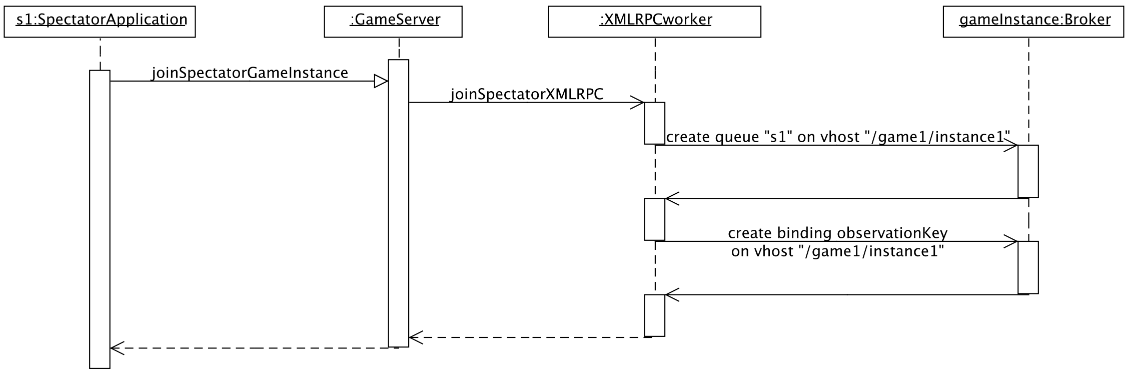
s1:SpectatorApplication
:GameServer
:XMLRPCworker
gameInstance:Broker
joinSpectatorGameInstance
joinSpectatorXMLRPC
create queue "s1" on vhost "/game1/instance1"
create binding observationKey
on vhost "/game1/instance1"

Figure 17: Sequence diagram of the joining of the Spectator Application to the game instance

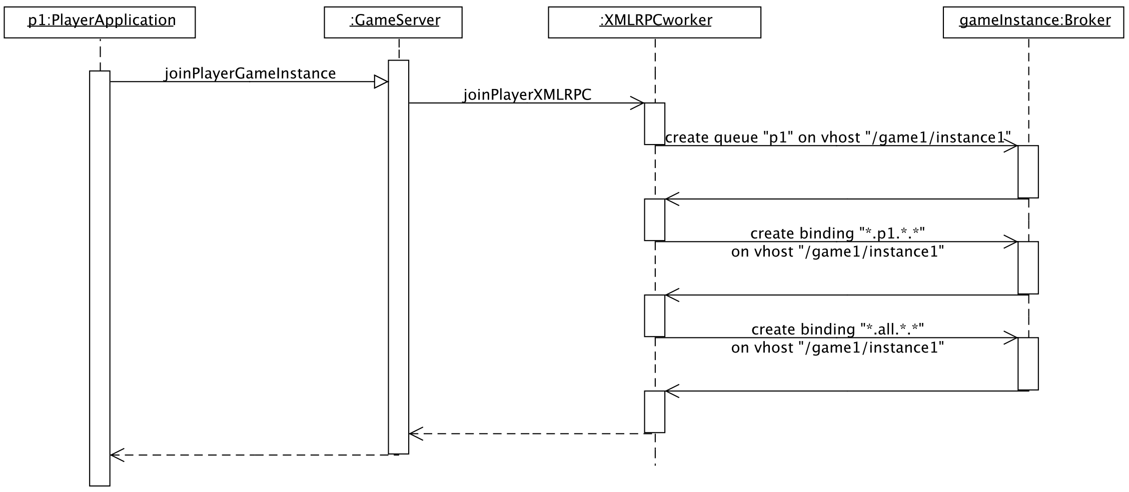
p1:PlayerApplication
:GameServer
:XMLRPCworker
gameInstance:Broker
joinPlayerGameInstance
joinPlayerXMLRPC
create queue "p1" on vhost "/game1/instance1"
create binding "*.p1.*.*"
on vhost "/game1/instance1"
create binding "*.all.*.*"
on vhost "/game1/instance1"

Figure 18: Sequence diagram of the joining of a Player Application to the game instance

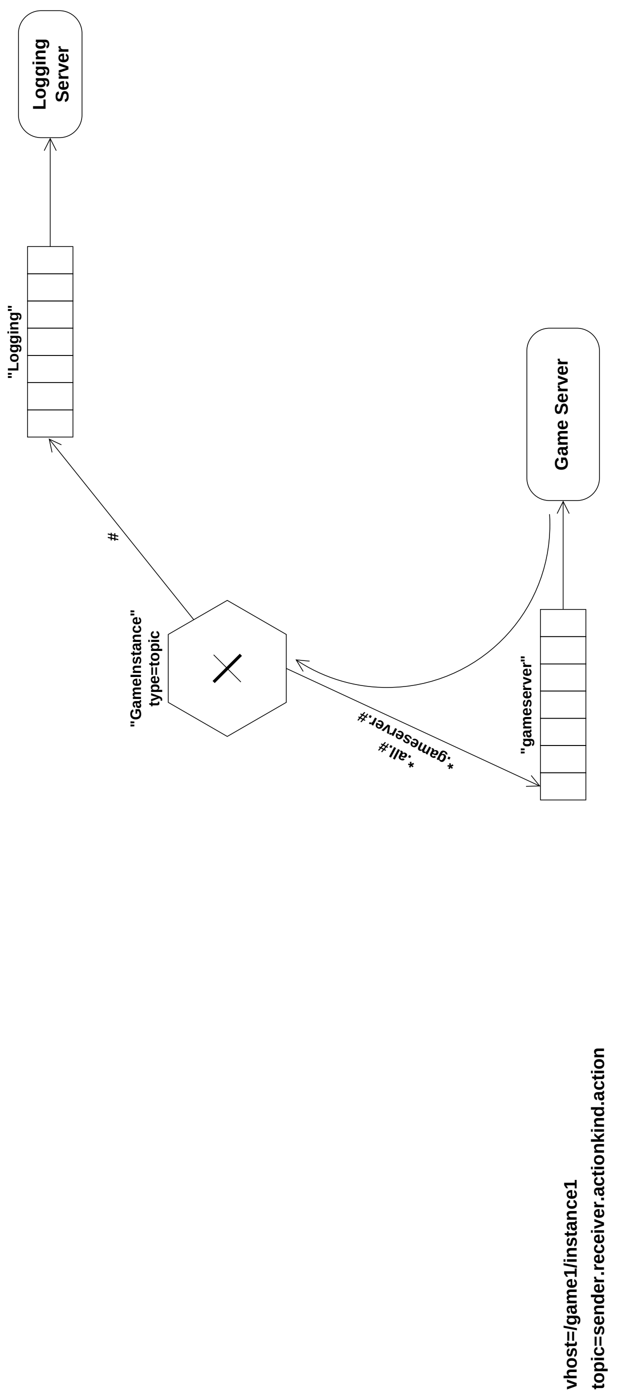

Figure 19: Architecture of the subsystem for the game instance (at creation time)

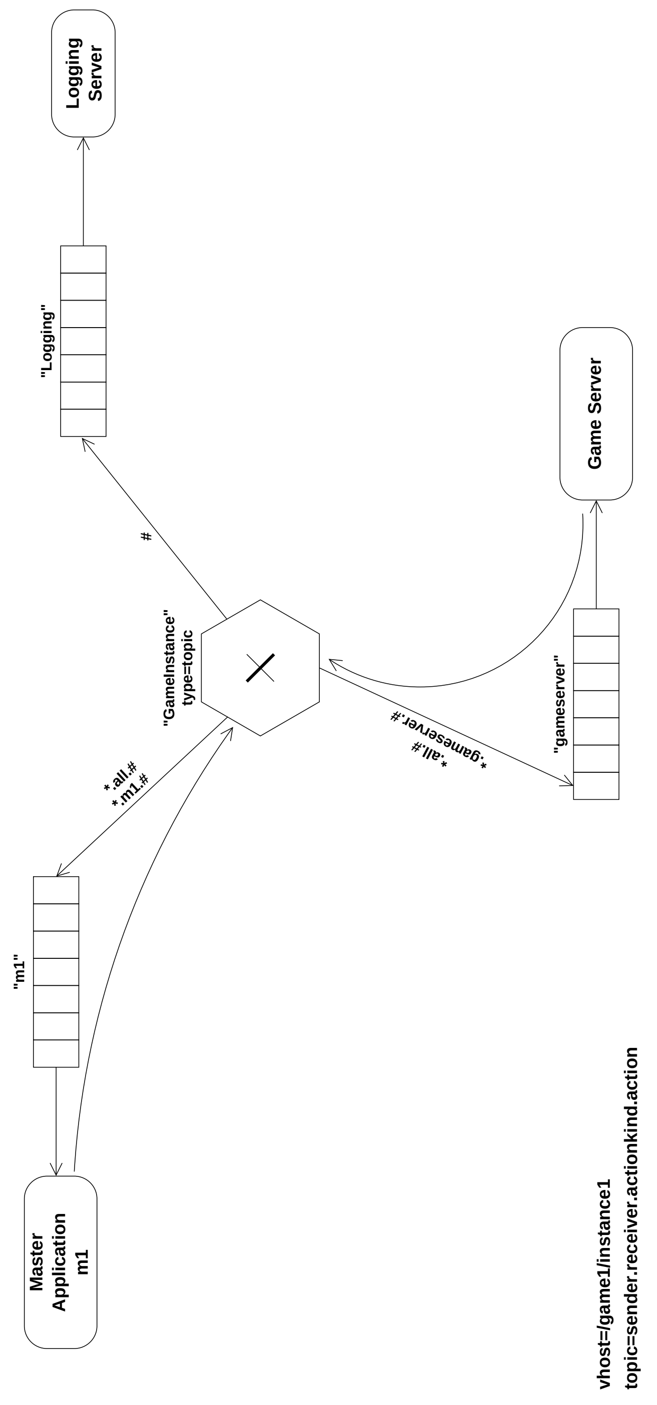

Figure 20: Architecture of the subsystem for the game instance after the game master has joined the game instance

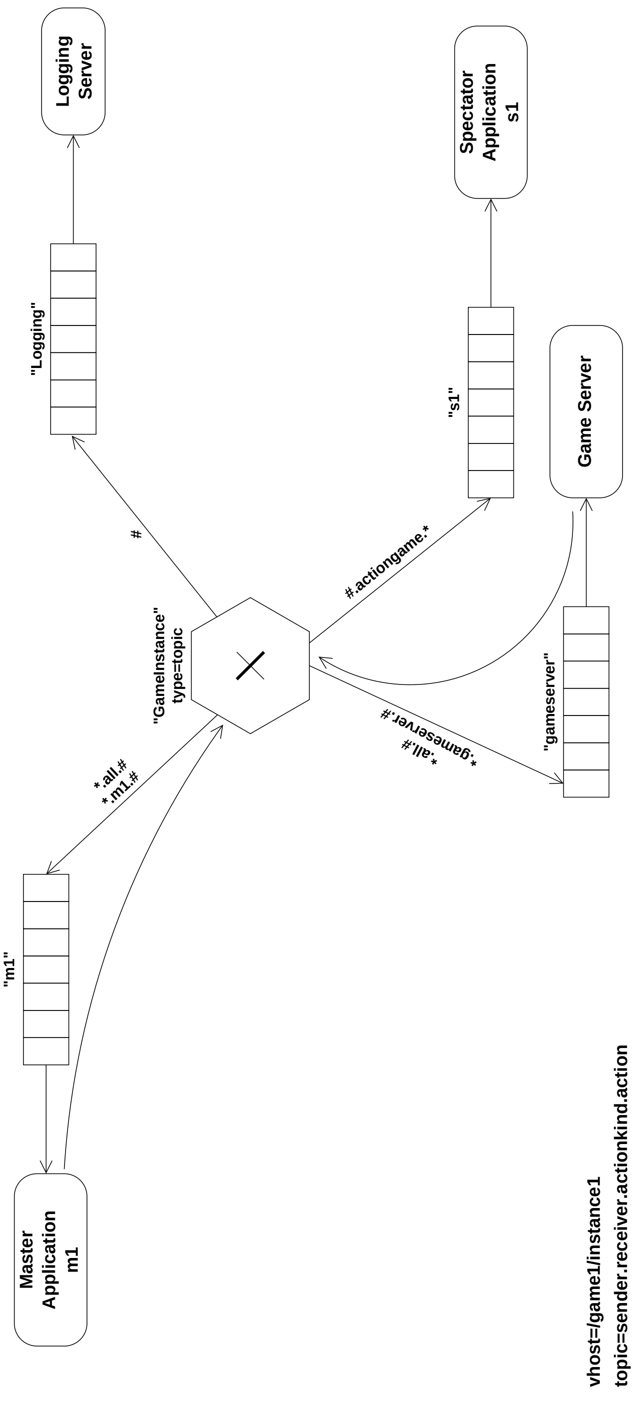

Figure 21: Architecture of the subsystem for the game instance after a spectator has joined the game instance

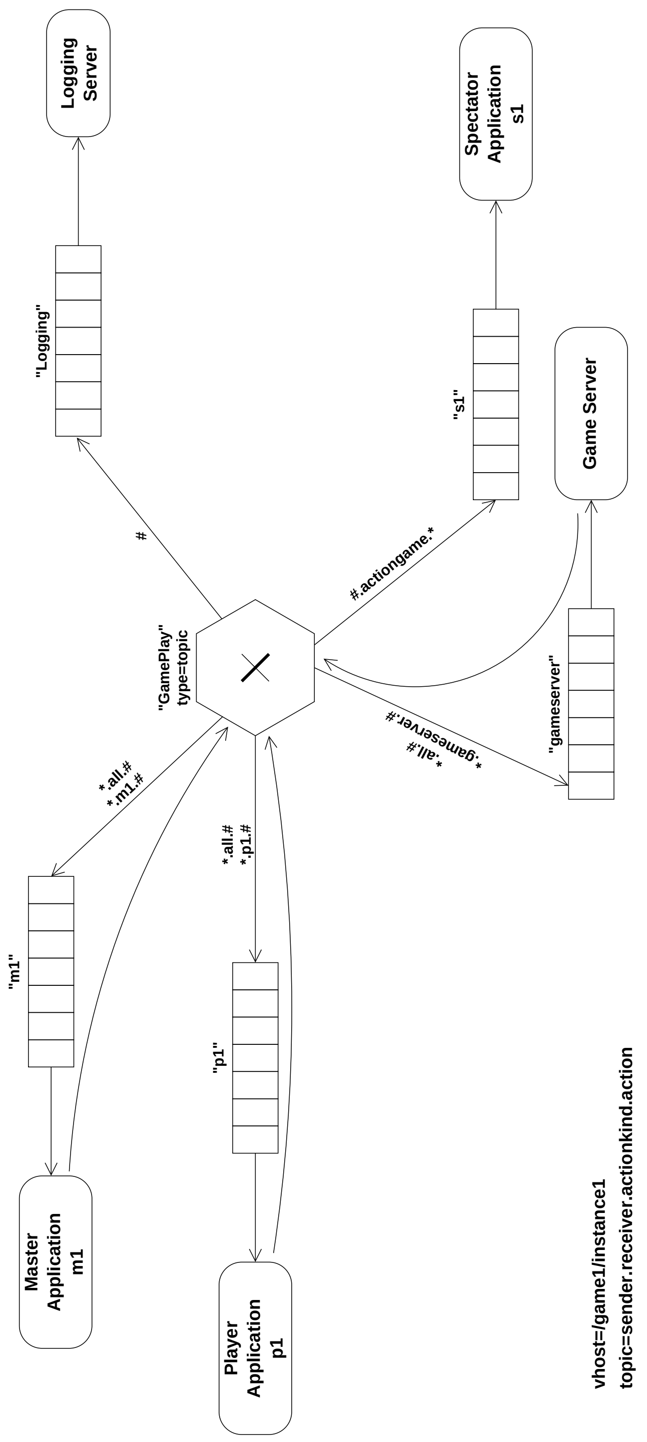

Figure 22: Architecture of the subsystem for the game instance after a player has joined the game instance

10.3 Implementation and execution

The implementation of the example application is available in the Subversion repository of the TOTEM Redmine in the directory structure `TOTEM.CommunicationMiddleware/Sources/-IntegrationExampleApplication`.

The prerequisites are listed in the file `readme.txt`:

- Software to install and to configure:
  - For the server side:
    - `Erlang` version $\geq$ R13B03,
    - `RabbitMQ Server` version $\geq$ 2.7.1,
    - `Pika` version $\geq$ 0.9.5.
  - For Android clients:
    - `Android SDK API level` $\geq$ 7.
  - For Java clients:
    - `Maven` version $\geq$ 2.2.1,
    - `Java JRE` $\geq$ 1.5.
  - For JavaScript clients:
    - `Node.js` $\geq$ 0.4.10
- Subsystems of the example application to "install":
  - GameServer: a Python project using XML-RPC communication to create and control the game logic servers. This is the project that should be integrated into the Web Framework.
  - GameLogicServer: a Python project using AMQP communication to control the execution of the game instance. This is where game developers put the logic of the server of the game.
  - MasterApplication for Java J2SE: a Maven project using the RabbitMQ Java Client (for J2SE[6]). The directory contains also the shell script `run.sh` to launch this subsystem.
  - PlayerMasterAndroid for Android device: This is the Eclipse Android project that can be imported into Eclipse for compiling and generating Android artefacts. When launching the application, it is possible to chose between creating or joining a game instance.
  - MasterApplicationJavascript: This is the Web application for the master. Before launching this application in a Web browser, you have to start NodeJsProxy first, as mentionned in NodeJsProxy/readme.txt.
  - SpectatorApplication for Java J2SE: a Maven project using the `RabbitMQ` Java Client (for J2SE). The directory contains also the shell script run.sh to launch this subsystem.
  - SpectatorApplicationJavascript: This is the Web application for spectators. Before launching this application in a Web browser, you have to start NodeJsProxy first, as mentionned in NodeJsProxy/readme.txt.
  - NodeJsProxy: This is the proxy used both by Master and Spectator applications in JavaScript to log-in and to establish their AMQP connections with the RabbitMQ broker.

[6]As demonstrated with the examples written using the RabbitMQ and XML-RPC Java Client for Android, there is no significant differences between a J2SE client application and its Android version.

  - PlayerApplication for Java J2SE: a Maven project using the `RabbitMQ` Java Client (for J2SE). The directory contains also the shell script run.sh to launch this subsystem.

NB: All the projects / directories mentionned in the previous list are complemented with `readme.txt` files. Please refer to the `readme.txt` file at the root directory for installation and execution instructions.

Important ►Do not forget to adapt the RabbitMQ and XML-RPC configuration properties such as the IP addresses of the RabbitMQ broker and XML-RPC servers to your execution environment. Here are the files you need to modify:

- For the Android Application: `PlayerMasterAndroid/res/raw/rabbitmq.properties` and `PlayerMasterAndroid/res/raw/xmlrpc.properties`
- For Java J2SE Applications: `*src/main/ressources/rabbitmq.properties` and `*src/main/resources/xmlrpc.properties`
- For Javascript Applications: `NodeJsProxy/resources/rabbitmq.properties` and `NodeJsProxy/resources/xmlrpc.properties`

◄

To launch the example application on a Unix-like operating system with the Java J2SE clients, execute the script ./run.sh. This script launches all the subsystems in sequence with some of the processes in background mode. The sequence starts with the stopping and initialisation of the RabbitMQ broker, so that no interference can happen with another running application.

Other scripts exist:

- for Unix-like operating systems:
  - run_with_android_phones.sh: the same as run.sh one but for running on Android phones.
  - run_with_master_and_spectators_javascript.sh: the same as run.sh but using Master and Spectator Web applications. Of course, you can also launch additional Java or Android players, and Java or Android spectators.
  - termination.sh: to send a terminate message to all the clients of all the available game instances. It terminates all the game instances and the whole server side.
- for Windows operating systems:
  - run_with_android_phones.bat: the same as run_with_android_phones.sh.
  - termination.bat: the same as termination.sh.

To run all these scripts, follow the instructions.

A Installation of RabbitMQ

In this section, we provide the procedure for installing RabbitMQ: the broker, the Java client for Android phones, the Python client for the web framework, and the JavaScript client for the Browser. Similar instructions are included in a separate section, namely Section B.2, for installing RabbitMQ on Amazon EC2 cloud.

A.1 RabbitMQ broker on desktop computers

In this section, we install the software from a regular archive so that we will execute RabbitMQ broker as a non-`root` user. The configuration of the environment variables `RABBITMQ_MNESIA_BASE` and `RABBITMQ_LOG_BASE` is important to allow executing the broker as a non-`root` user. This is done as follows.

1. The instructions we provide are extracted from the "install" Web page at this URL: `http://www.rabbitmq.com/install.html`. Refer to this Web page when necessary.

   - At first, install Erlang and Python 2.6:
     - On GNU/Linux, Debian distribution, check the versions that are installed, executing the following commands:
       ```
       $ erl
       Erlang R13B03 (erts-5.7.4) [source] [smp:2:2] [rq:2] [async-threads:0] [hipe] [kernel-p

       Eshell V5.7.4  (abort with ^G)
       ...
       $ python --version
       Python 2.6.6
       ```
     - Or execute the following commands as the user root or as a sudouser:
       ```
       apt-get install erlang
       apt-get install python
       ```
   - Next, install RabbitMQ version 2.7.1 from the "Package for generic Unix systems" (file rabbitmq-server-generic-unix-2.7.1.tar.gz) or from the "Windows Bundle" (file rabbitmq-server-windows-2.7.1.zip). In the following, adapt the commands if you are running a Windows operationg system (cf. the Web page `http://www.rabbitmq.com/-install.html#windows`).
     - On GNU/Linux, Debian distribution, execute the following commands:
       ```
       wget http://www.rabbitmq.com/releases/rabbitmq-server/ \
            v2.7.1/rabbitmq-server-generic-unix-2.7.1.tar.gz
       tar xfz rabbitmq-server-generic-unix-2.7.1.tar.gz
       ```

   On Unix-like operating systems, the broker of RabbitMQ, called the server in RabbitMQ terminology, is now installed in the directory ${YOURDIRECTORY}/rabbitmq_server-2.7.1. The commands of RabbitMQ (rabbitmq-server to launch the broker and rabbitmqctl to control the broker) are available in the directory ${YOURDIRECTORY}/rabbitmq_server-2.7.1/sbin.

2. (a) On Unix-like operating systems, in order to add the RabbitMQ commands to the shell path, and to configure the location of the database and the log files, add the following commands to the shell script that is executed when opening a shell connection, for instance in the file ~/.bashrc:

   ```
   PATH=$PATH:${YOURDIRECTORY}/rabbitmq_server-2.7.1/sbin
   export RABBITMQ_MNESIA_BASE=${YOURDIRECTORY}/rabbitmq_server-2.7.1/mnesia
   export RABBITMQ_LOG_BASE=${YOURDIRECTORY}/rabbitmq_server-2.7.1/log
   ```

Note that we assume that you are now going to create the directories ${YOURDIRECTORY}/rabbitmq_server-2.7.1/mnesia and ${YOURDIRECTORY}/rabbitmq_server-2.7.1/log.

(b) On Windows systems, follow the instructions in `http://www.rabbitmq.com/install.html#windows`, that is:

- Set ERLANG_HOME environment variable to where you actually put your Erlang installation, e.g. C:\Program Files\erl5.7.4 (full path). The RabbitMQ batch files expect to execute %ERLANG_HOME%\bin\erl.exe.
- Create a system environment variable (e.g. RABBITMQ_SERVER) for C:\Program Files\RabbitMQ\rabbitmq_server-2.7.1. Adjust this if you put rabbitmq_server-2.7.1 elsewhere.
- Append the literal string `;%RABBITMQ_SERVER%\sbin` to your system path (as known as `%PATH%`).

Note: On Windows systems, it is not necessary to create and set the variables `RABBITMQ_MNESIA_BASE` and `RABBITMQ_LOG_BASE`.

3. On Unix-like and Windows operating systems, your RabbitMQ installation can be tested by launching the broker as follows:

```
$ rabbitmq-server -detached # execute in the background
Activating RabbitMQ plugins ...
0 plugins activated:
$ rabbitmqctl status
Status of node rabbit@rabbit ...
[running_applications,[rabbit,"RabbitMQ","2.7.1",
                        mnesia,"MNESIA  CXC 138 12","4.4.14",
                        os_mon,"CPO  CXC 138 46","2.2.5",
                        sasl,"SASL  CXC 138 11","2.1.9.2",
                        stdlib,"ERTS  CXC 138 10","1.17",
                        kernel,"ERTS  CXC 138 10","2.14"],
 nodes,[disc,[rabbit@rabbit]],
 running_nodes,[rabbit@rabbit]]
...done.
$ rabbitmqctl stop
Stopping and halting node rabbit@rabbit ...
...done.
$ rabbitmqctl status
Status of node rabbit@rabbit ...
Error: unable to connect to node rabbit@rabbit: nodedown
diagnostics:
- nodes and their ports on rabbit: [rabbitmqctl2776,45667]
- current node: rabbitmqctl2776@rabbit
- current node home dir: /home/bitnami
- current node cookie hash: 58UmUjslvHMdJuJyRDcEag==
```

A.2 RabbitMQ producers and consumers on J2SE applications, dedicated RabbitMQ Java client library

The library for Java producers and consumers of RabbitMQ, called clients in RabbitMQ terminology, is downloaded using Maven. To test the library with your environment, you can do the following:

- At first, install Sun JDK 1.6 and Maven 2:
  - On GNU/Linux, Debian distribution, check the versions that are installed, executing the following commands:

```
$ java -version
java version "1.6.0_24"
...
$ javac -version
javac 1.6.0_24
$ mvn -version
Apache Maven 2.2.1 (rdebian-4)
Java version: 1.6.0_24
Java home: /usr/lib/jvm/java-6-sun-1.6.0.24/jre
```

   – Or execute the following commands as the user root or as a sudoer:

```
apt-get install sun-java6-jdk
apt-get install maven2
```

- Download the directory `Totem.CommunicationMiddleware/Sources/TutorialExamples/J2SE/Step1` from the TOTEM Redmine Subversion repository.
- Launch the RabbitMQ broker, and the J2SE consumers and producers using the script run.sh.

A.3 RabbitMQ producers and consumers on desktop computers, Python RabbitMQ client library named Pika

To limit the number of commands requiring `root` privileges, we propose to not install the Python packages using the `pip` or `easy_install` utility tools. But, of course, you can prefer performing this installation differently using Python utility tools. In addition, we only provide the commands for Unix-like operating systems.

1. Install the software Pika, version 0.9.5 by executing the following commands

```
$ cd ${YOURDIRECTORY}
$ wget http://pypi.python.org/packages/source/p/pika/pika-0.9.5.tar.gz
$ tar xfz pika-0.9.5.tar.gz
```

   The Python packages Pika is installed in the directory ${YOURDIRECTORY}/pika-v0.9.5. Insert this package in the path of Python by adding the directories in the shell variable PYTHONPATH:

```
export PYTHONPATH=$PYTHONPATH:${YOURDIRECTORY}/pika-v0.9.5
```

2. Check that Pika is correctly installed by trying the following Python commands that demonstrate that the Pika package can be imported in a Python script:

```
$ python
Python 2.6.5 (r265:79063, Apr 16 2010, 13:09:56)
[GCC 4.4.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import pika
>>> quit()
```

To test the library with your environment, you can do the following:

- Download the directory `Totem.CommunicationMiddleware/Sources/TutorialExamples/Pika/Step5` from the TOTEM Redmine Subversion repository.
- Start the RabbitMQ server with the following command:

```
$ rabbitmq-server
Activating RabbitMQ plugins ...
0 plugins activated:
...
```

- Check that there is no exhange named topic_logs by executing the following commands:

```
$ rabbitmqctl list_exchanges
Listing exchanges ...
amq.direct direct
amq.topic topic
amq.rabbitmq.log topic
amq.fanout fanout
amq.headers headers
direct
amq.match headers
...done.
$ rabbitmqctl list_bindings
Listing bindings ...
...done.
```

- Launch the consumers and the producers using the script `run.sh`.

A.4 RabbitMQ JavaScript producers and consumers using JavaScript AMQP library

The library for Javascript AMQP producers and consumers works with Node.js, which is an event-driven I/O server-side JavaScript environment. Here follows an installation procedure for Unix-like operating systems which should be performed on the server side:

1. Install Node.js:

   - The prerequisites to build Node.js from source are:
     - python: version 2.4 or higher. The build tools distributed with Node.js run on python.
     - libssl-dev: If you plan to use SSL/TLS encryption in your networking, you need this library. `libssl` is the library used in the `openssl` tool.
   - To download and build Node.js, version 0.4.10 execute the following commands:

   ```
   $ mkdir ~/node.js
   $ cd  ~/node.js
   $ wget http://nodejs.org/dist/node-v0.4.10.tar.gz
   $ tar xfz node-v0.4.10.tar.gz
   $ rm node-v0.4.10.tar.gz
   $ cd node-v0.4.10
   $ ./configure --prefix=$HOME/node.js/node-v0.4.10
   $ make
   $ make install
   $ echo "export PATH=$HOME/node.js/node-v0.4.10/bin:$PATH" >> ~/.bashrc
   $ echo "export NODE_PATH=$HOME/node.js/node-v0.4.10:$HOME/\
     node.js/node-v0.4.10/lib/node_modules" >> ~/.bashrc
   $ source ~/.bashrc
   ```

   - Check that node.js is properly installed by trying the following command:

   ```
   $ node -v
   v0.4.10
   ```

2. Install npm, the package manager for installing additional Node.js libraries:
   - If the curl command is not present, you should first install it:
     ```
     $ sudo apt-get install curl
     ```
   - Then, install npm:
     ```
     $ curl http://npmjs.org/install.sh | sh
     ```
3. Install node-amqp *(version 0.1.0)*, an AMQP client for Node.js by executing the following command:
   ```
   $ npm install -g amqp@0.1.0
   ```
4. Install node-mxlrpc, an XMLRPC client for Node.js by executing the following command:
   ```
   $ npm install -g xmlrpc
   ```

To test the AMQP library with your environment, you can do the following:

- Download the directory `Totem.CommunicationMiddleware/Sources/TutorialExamples/Javascript` from the TOTEM Redmine Subversion repository.
- Enter the `Step1` folder, and launch the consumer and the producer using `run.sh`.
- At the end of the execution of the script `run.sh` , check that the output looks like this one:

  ```
  [...]
  Starting node rabbit@rabbit ...
  ...done.
  Producer connected to broker.
  queue declared.
  message published.
  Listing queues ...
  hello_queue false false
  ...done.
  Consumer connected to broker.
  Message received: Hello world!
  Stopping and halting node rabbit@rabbit ...
  ...done.
  END OF HELLOWORLD TUTORIAL TEST
  ```

  Warning: ►Using the script to launch the consumer and the producer will delete every queues and exchanges located on your broker. If you don't want to do so, please refer to the `readme.txt` file.◄

B Installation of RabbitMQ and Pika on Amazon EC2

Important note: ▶This appendix is deprecated since we have not done recent tests on the Amazon Cloud Computing Platform EC2. Please ask if you need an update of this section.◀

In this section, we explain how to launch and configure an EC2 instance for executing the RabbitMQ broker (in Erlang), and some Pika producers and consumers (in Python). The steps presented in this section could be used to create a dedicated EC2 "Amazone Machine Image" (AMI), as named in the Amazone EC2 vocabulary, for TOTEM. Similar instructions are included in a separate section (namely Section A) for installing RabbitMQ on a desktop computer.

B.1 Create an "Instance"

First of all, create an account on Amazon Elastic Compute Cloud (EC2): `https://aws.amazon.-com`. Then, connect to the "AWS Management Console" and create an instance as follows:

1. Choose the EC2 page.
2. Select an "Amazon Machine Image" by "Viewing" all images of "All platforms" with for example the identifier `ami-06f80e6f`. This image is an Ubuntu operating system version 10.4.
3. Launch an image using the contextual menu of the AMI you previously selected (by right clicking). In the "Request Instances Wizard", choose an "Instance type" of type Micro for instance: it is sufficient for our tests. At the "Create Key Pair" step, create a new key pair and do not forget to load it in order to be granted an access through a SSH connection. Be careful! If you loose this key, you somewhat loose the instance you have just created. The saved file (*.pem) must be put in a directory with restricted access (usely named ~/.ssh on a Unix-like system, and with the rights 0700) and must be assigned restricted rights (0400). At the "Configure Firewall" step, create a new security group. At the end of this procedure, the instance is running: Go to see the "Instances" by selecting the entry Instances in the menu on the left.
4. Browse the contextual menu of the instance that is running and have a look at the "Instance Management" menu and the "Instance Lifecycle" menu.
5. When an instance is selected, information such as the "Public DNS" are provided in the frame under the list of instances. A "Public DNS" is an IP address such as for example `ec2-174-129-122-66.compute-1.amazonaws.com`

B.2 Configure the "Instance"

The next step is to connect to the instance and install the software for the example application. This is done as follows:

1. For the instance you have created following the steps of the previous section, open the TCP port number 22 for the procotol SSH. Go to the "Security Groups" frame by selecting the corresponding entry in the menu on the left. Select the security group you have previously created and add a new "Allowed Connection" of "Connection Method" type SSH and "Source" value IP_address_of_your_computer/0. The other information such as "From Port" and "To Port" are correctly configured to the default value 22.
2. You can now open a shell connection to the instance using the following command (by adapting it to your use case):

```
$ ssh -i ${KeyPairFile.pem} bitnami@${PublicDNS}
```

Note that you need the "Key Pair" file and the "Public DNS". The login name bitnami is a default login name provided by the AMI providers. This user is a "sudoer" and can thus use the command `sudo` for executing a command as the user `root` when necessary.

3. Install the Ubuntu package for Erlang with all its dependencies using the utility tool `apt-get`:

```
sudo apt-get install erlang
```

4. Install the software RabbitMQ, version 2.7.1, by using the following commands:

```
$ wget http://www.rabbitmq.com/releases/rabbitmq-server/v2.7.1/\
  rabbitmq-server-generic-unix-2.7.1.tar.gz
$ tar xfz rabbitmq-server-generic-unix-2.7.1.tar.gz
```

Note that we do not install the Ubuntu package of RabbitMQ, but rather prefer installing the software from a regular Unix archive so that we will execute RabbitMQ as a non-`root` user. The broker of RabbitMQ, called the server in RabbitMQ terminology, is now installed in the directory /home/bitnami/rabbitmq_server-2.7.1. The commands of RabbitMQ (rabbitmq-server to launch the broker and rabbitmqctl to control the broker) are available in the directory /home/bitnami/rabbitmq_server-2.7.1/sbin. In order to add the RabbitMQ commands to the shell path, and to configure the location of the database and the log files, execute the following commands:

```
$ echo "PATH=$PATH:rabbitmq_server-2.7.1/sbin" >> .bashrc
$ mkdir /home/bitnami/rabbitmq_server-2.7.1/mnesia
$ echo "export RABBITMQ_MNESIA_BASE=/home/bitnami/rabbitmq_server-2.7.1/mnesia" >> .bashrc
$ mkdir /home/bitnami/rabbitmq_server-2.7.1/log
$ echo "export RABBITMQ_LOG_BASE=/home/bitnami/rabbitmq_server-2.7.1/log" >> .bashrc
$ . .bashrc # to update the environment of the current shell connection
```

Note that RabbitMQ environment variables default to the values /var/lib/rabbitmq/mnesia and /var/log/rabbitmq, preventing the launching of the broker as a non-`root` user. We terminate the installation of RabbitMQ by following the instructions provided in the section "Issues with hostname" of the Web page http://www.rabbitmq.com/ec2.html and we apply the following commands:

```
sudo -s
echo "rabbit" > /etc/hostname
echo "127.0.0.1 rabbit" >> /etc/hosts
hostname -F /etc/hostname
exit
```

5. Your RabbitMQ installation can be tested by launching the broker as follows:

```
$ rabbitmq-server -detached # execute in the background
Activating RabbitMQ plugins ...
0 plugins activated:
$ rabbitmqctl status
Status of node rabbit@rabbit ...
[running_applications,[rabbit,"RabbitMQ","2.7.1",
                        mnesia,"MNESIA  CXC 138 12","4.4.14",
                        os_mon,"CPO  CXC 138 46","2.2.5",
                        sasl,"SASL  CXC 138 11","2.1.9.2",
                        stdlib,"ERTS  CXC 138 10","1.17",
                        kernel,"ERTS  CXC 138 10","2.14"],
 nodes,[disc,[rabbit@rabbit]],
 running_nodes,[rabbit@rabbit]]
...done.
$ rabbitmqctl stop
Stopping and halting node rabbit@rabbit ...
...done.
$ rabbitmqctl status
Status of node rabbit@rabbit ...
Error: unable to connect to node rabbit@rabbit: nodedown
diagnostics:
- nodes and their ports on rabbit: [rabbitmqctl2776,45667]
- current node: rabbitmqctl2776@rabbit
- current node home dir: /home/bitnami
- current node cookie hash: 58UmUjslvHMdJuJyRDcEag==
$ jobs
```

6. In order to allow communication with the RabbitMQ broker installed on the cloud from your computer, add a new "Allowed Connection" to your security group with the following parameters: "Connection Method" is Custom, "Protocol" is TCP, "From Port" and "To Port" are 5672 (default port number of RabbitMQ broker), "Source IP" is IP_address_of_your_computer/0.

7. Install the software Pika, version 0.9.5 by executing the following commands

```
$ wget http://pypi.python.org/packages/source/p/pika/pika-v0.9.5.tar.gz
$ tar xfz pika-v0.9.5.tar.gz
```

   Note that we do not install the Python packages using the `pip` or `easy_install` utility tools; you can thus perform this step differently. The Python package Pika is installed in the directories /home/bitnami/pika-v0.9.5. Insert this package in the path of Python by adding the directories in the shell variable PYTHONPATH:

```
$ echo "export PYTHONPATH=/home/bitnami/pika-v0.9.5" >> .bashrc
```

8. Check that Pika is correctly installed by trying the following Python commands that demonstrate that the Pika package can be imported in a Python script:

```
$ python
Python 2.6.5 (r265:79063, Apr 16 2010, 13:09:56)
[GCC 4.4.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import pika
>>> quit()
```

B.3 Create and share a TOTEM "Amazon Machine Image"

To be decided if this is the role of the project. We know how to do it.

References

[AMQP Consortium, 2008] AMQP Consortium (2008). AMQP: Advanced Message Queuing Protocol, Version 0-9-1. Protocol specification, AMQP Consortium.

[Mühl et al., 2006] Mühl, G., Fiege, L., and Pietzuch, P. (2006). *Distributed Event-Based Systems.* Springer.

[Videla and Williams, 2011] Videla, A. and Williams, J. (2011). *RabbitMQ in Action.* Manning.