Gaming Platform Version 1.0

Candidate Version 12-June-2003

Open Mobile Alliance
OMA-GamingPlatform-V1_0-20030612-C

Continues the Technical Activities Originated
in the Mobile Games Interoperability Forum

Contents

1. Scope

The potential scope of a gaming platform is enormous. A pragmatic standpoint has, therefore, been taken, where initial efforts have been concentrated in those areas that are deemed to reap most benefit for the game developer. Specifically, in the first release of the API, the following areas are addressed:

- Session management: provides the identifiers that bind the user interactions into single concept of a game, provides access to the other APIs and provides the interface through which the lifecycle of game entities can be managed.
  Rationale: the core framework upon which all other API access is built.

- Connectivity: provides the communication layers, protecting the developer from the low-level implementation details of the transport mechanism.
  Rationale: network access is widely reported to cause significant rework on the part of the game developer.

- Metering: provides a standard API through which the game can inform the gamingplatform of game specific billable events.
  Rationale: relates fundamentally to how the game is paid for and so of high importance.

- Scores and Competition Management: provides the mechanisms by which the game can report and retrieve scores from the gaming platform, so allowing competitions to be run in a unified manner.
  Rationale: the basis upon which online communities can be built in the mobile gaming arena.

- Logging: provides a standard reporting mechanism by which a game informs the gaming platform of its status. This insulates against specific formatting requirements and through the implementation of variable logging levels, assists in the troubleshooting process.
  Rationale: by standardizing logging troubleshooting is simplified and thus operational costs reduced.

- Timers: provides the mechanism by which a game schedules and delays activities.
  Rationale: provides unified access to time based event triggers for the game developer.

Underlying the design of the APIs discussed in this document lays an event-based mechanism. The Session API defines the core of the event model. Although not necessary, a familiarity of event based programming will significantly help in the interpretation of this document.

The framework of APIs offers no guarantees on the re-entrancy of the event handlers. Specific game platform vendors may offer tools and reentrance conditions on top of the APIs, but this is an implementation issue, and out of the scope of this specification.

2. References

2.1 Normative References

[CREQ] "Specification of WAP Conformance Requirements". Open Mobile Alliance™. WAP-221-CREQ. URL:http//www.wapforum.org/ *<to be replaced by an OMA ref when available>*

[RFC2119] "Key words for use in RFCs to Indicate Requirement Levels". S. Bradner. March 1997. URL:http://www.ietf.org/rfc/rfc2119.txt

[RFC2234] "Augmented BNF for Syntax Specifications: ABNF". D. Crocker, Ed., P. Overell. November 1997. URL:http://www.ietf.org/rfc/rfc2234.txt

[OMAGPJD10] OMA-GamingPlatform-JavaDocs-V1_0-20030525-D

2.2 Informative References

No references.

3. Terminology and Conventions

3.1 Conventions

The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” in this document are to be interpreted as described in [RFC2119].

All sections and appendixes, except “Scope” and “Introduction”, are normative, unless they are explicitly indicated to be informative.

3.2 Definitions

Actor	Represents the persistent data for the ActorSession.
ActorEvent	Event generated by a user.
ActorSession	Representation of a User in an ApplicationInstance. One user can have many Actors.
Administration	The task of modifying the behaviour of the product after it has been released and is in operation. The actor is usually other than System Integrator or Developer e.g. Administrator or Customer using administration tools.
Agent	A computer guided user of the service.
Application Context	The application context API provides the application developer with information regarding the invocation, e.g. the current player, the incoming message, the triggered event, the incoming HTTP request etc.
Application pull	Service type where an application gets data by requesting it from a mobile terminal.
Application push	Service type where an application sends data to a mobile terminal.
Application server / Core platform	Server-side software platform that provides network connectivity and general support for various applications. The application server may handle e.g. user sessions and load balancing. More specific platforms (e.g. a game platform) can be constructed on top of the application server software.
ApplicationEvent	Event generated by an application.
ApplicationInstance	An object representing an instance of an application. It can have any number of Actors.
Asynchronous	Not synchronized; that is, not occurring at predetermined or regular intervals. The term asynchronous is usually used to describe communications in which data can be transmitted intermittently rather than in a steady stream. This term relates to electronic communication, where participants send messages to others for reading at another time.
Avatar	User ‘character’. A game can use avatar information when creating the player. Avatar can contain e.g. scores, skins and results.
Billing	To aggregate and process charging events in order to effect settlement through a financial transaction.

Binary portability
Binary portability makes it possible to move an executable copy of a program from one machine to another without any modification. Using an execution environment like Java, the virtual machine provides binary portability across platforms.

Browser client
A terminal that contains software that allows a user to view or "browse" text-based or multimedia information on the Internet.

Competition
An event where Users compete against each other or try to reach a set objective. Competitions may last for a defined period of time. Usually some sort of prizes are awarded for the competitors based on their Scores.

Configuration
The task of modifying the behaviour of the product as a separate task from programming. Configurations are done as part of the development cycle, integration and administration. The behaviour of the product is a very wide-ranging term covering: user interface, logic, storage of data, performance, session management, logging etc.

Content development
The part of the Development work that produces the Service's content, for example question banks or game worlds. Content development includes localisation of the content.

Contest
- Contest is a structural description of an entertainment concept involving one or more games
- Contests have a time frame when they are active i.e. playable, when game lists are accessed via contests, only active games are shown
- Contests may have rules that control who can participate or win, when game lists are accessed via contests, only accessible games are shown to a particular player
- Contests may consist of phases that are contests
- Contests involve an award that may be a qualification for the next phase
- Regarding reference, description and management contests are similar to games

Customer
The organization providing the product for the End-Users. Customisation is done according to the Customer requirements. For products offering mobile services, the Customer is often a teleoperator or a portal owner.

Customisation
The task of modifying the behaviour and appearance of the product due to specific customer needs. This is usually achieved by Configuring the product.

Customised product
A product generated and configured according to Customer's wishes.

Data administration
The part of Administration, which is related to administrating the data entities the product uses. E.g.: updating content files and templates, game configuration, user database and the set of playable games.

Declarative Events
Application events that can be declared (scheduled) to occur in a reoccurring fashion.

Delivery report
A message notifying the sender whether the recipient has received a previously sent message.

Design portability
Design portability is the ability to create applications, which only rely on the knowledge of objects interfaces. As an example, Object Management Group's (OMG) CORBA architecture defines language independent IDL interfaces that facilitate design portability.

Development
The task of producing all necessary components of a product. The tasks include e.g. programming, initial Configuration, content development and user interface development. Development precedes Integration. Development work should support Integration and Administration for example by configurability.

Dynamic configuration
Dynamic configuration is the part of Configuration work that can be done to software that is running without stopping the software or the on-going sessions. This feature helps the configuration of a released product. Examples of Dynamic configuration: tuning the product

performance, adding new features or correcting bugs.

End-user

The real human user using the service, i.e. playing a game.

Event

An action or occurrence detected by a program. Events can be user actions, such as clicking a mouse button or pressing a key, or system occurrences, such as running out of memory. Most modern applications, particularly those that run in Macintosh and Windows environments, are said to be event-driven, because they are designed to respond to events.

Event-based programming

A method of programming in which blocks of code are run as users do things or as events occur while a program is running.

Executable Client

The executable client has local processing and storage capacity, e.g. J2ME clients. Executable clients facilitate applications that can be used also when the terminal is not connected to the network or out of coverage.

Game category

Similar kinds of Game Concepts form a Game category, e.g. trivia games, adventure games and board games. The games that belong to the same category have many similarities. Therefore the same design work and implemented components can often be utilised. There is no clear division of games to game categories and some categories overlap. One game may belong to many categories.

Game concept

Similar kinds of games form a Game concept. E.g. all customised and localised versions of Monopoly are part of Monopoly game concept. Often the different versions are implemented by Configuring one base implementation.

Game Execution Environment (GEE)

The Game Execution Environment is an execution environment for game applications using an OMA compliant gaming platform.

HTTP Connector

The HTTP Connector hides the intricacies of the HTTP protocol. The connector creates objects that encapsulate the HTTP request information.

Integration

The task of making a product ready for Production. This includes integrating the work done in software, user interface and content development. There is often also need for Configuration of the product according to customer and locale needs.

Interoperability

Interoperability refers to the capability for applications running on different computers to exchange information and operate cooperatively using this information. In other words, the ability to share data and services.

Localisation

The task of modifying the behaviour and appearance of the product due to the culture and language of its end-users. This is usually achieved by Configuring the product.

Localised product

A product generated and configured according to locale requirements.

Maintenance

The task of making bug corrections and fine-tunings to a product that is in production.

Master ApplicationInstance

An object representing the deployed, static parts of the application (game).

Message API

Contains interfaces to handle message-based connections like SMS, EMS, MMS and e-mail.

Message client

A terminal that contains software that allows a user to send and receive messages.

Message Connector

The Message Connector hides the intricacies of different messaging protocol, e.g. SMS (via CIMD2, UCP, SMPP etc), EMS, MMS, Nokia Smart Messaging, WAP Push etc. The Message Connector creates objects that encapsulate incoming message information.

OMA Binary Portable

Gaming platforms implementing the OMA defined game execution environment and services enabling portability of games. Binary port-ability requires selection of CPU and operating system, virtual machine or other binary execution environment, e.g. Win32, Java 2 Platform, Microsoft .NET CLR.

OMA Design Portable

Gaming platforms implementing the OMA defined services using native game execution environment. Games running on this platform only utilize standard OMA defined services and interfaces.

OMA Interoperable

Gaming platforms implementing native game execution environment providing OMA defined external interfaces that enable interoperability of different OMA compliant platforms.OMA Interoperable gaming platforms do not have to implement all the standard services and interfaces. Games developed forOMA Interoperable gaming platform are not portable.OMA Binary and Design portable platforms are also interoperable.

Mobile phone

A term often used interchangeably with cellular phone or wireless phone. Initially, a mobile phone referred to a phone attached to a vehicle, the vehicle's battery and had an external antenna. Mobile phones were distinguished from transport-able, portable, cordless and personal phones.

Mobile Phone

A term often used interchangeably with cellular phone or wireless phone. Initially, a mobile phone referred to a phone attached to a vehicle, the vehicle's battery and had an external antenna. Mobile phones were distinguished from transportable, portable, cordless and personal phones. See also Transportable Phones.

Operating environment

All the external software and hardware that are not part of the software itself, but may affect the software's behaviour. The server software's environment may include e.g. data connections, hardware, operating system, network services, database and monitoring software.

Operator logo

A small icon displayed on the screen of the mobile phone. The logo includes an identifier and thus can be assigned to a specific network operator. The logo will be displayed as long as the mobile phone is booked into the network of the corresponding operator.

Portability

Portability refers to the capability for software to run on any platform without modification. Portability means the ability of a game to run on any OMA compliant gaming platform.

Portal

A service that offers a broad array of resources and services, such as e-mail, forums, search engines, entertainment, and on-line shopping malls.

Post-paid

Calling plan where user gets a monthly bill.

Pre-paid

Calling plan where user must pay before he/she can make calls.

Programmatic Events

Events that can be added (scheduled) programmatically via the event API.

Protocol

An agreed-upon format for transmitting data between two devices.

Re-entrant

The attribute of a program or routine that allows the same copy of the program or routine to be used concurrently by two or more tasks.

Request/reply paradigm

A common pattern of communication used by application programs, where a client sends a request message to a server, the server responds with a reply message, and the client blocks waiting for this response.

Score
A single comparable integer value representing how well a user has done in the game.

Score Table
A list of Scores of Users organized by pre-defined criteria.

Session
A series of interactions encompassing the Actor’s lifetime within the specific Application Instance.

Software development
The part of Development work that produces the software of the product. This includes generation of new software components, i.e. programming, as well as selecting existing software components and versions to be used by the new software.

Source code portability
Source code portability requires a program to be recompiled when moving from one machine to another.

Subscriber Identity Module
Designed to be inserted into a mobile telephone, a SIM or "smart" card contains all subscriber-related data, such as phone numbers, service details, and memory for storing messages. With a SIM card, calls can be made from any valid mobile phone because the subscriber data--not the telephone's internal serial number--is used to make the call.

Synchronous
Occurring at regular intervals. The opposite of synchronous is asynchronous. This term relates to electronic communication, where participants communicate with each other at the same time.

System administration
The part of Administration, which is related to administrating the Operating environment in which the product is used, e.g. the operating system and network services.

Terminal/Access device
The device, e.g. a handset or a computer, which is used to communicate with the Server. Terminal manages the connection and communication between the End-user and the server. Terminal includes User agent software. Terminal is usually used to accomplish also other tasks than using games, e.g. to make phone calls or to use client side applications.

Tournament
An event where Users compete against each other within a common game.

Transaction
An atomic unit of work that modifies data. A transaction encloses one or more program statements, all of which either complete or roll back.

User
An object representing the actual end-user (that can be billed).

User agent
The software located at the Terminal that manages the communication between the End-user and the server. In browser enabled Terminals User agent is the browser. User agent is general-purpose software that is meant to allow the usage of all service types.

User agent:
- Manages the user interface and presents the game information to the End-user, e.g. texts, links, questions, game pieces and sounds
- Interprets the End-user commands, e.g. pressing a button
- Handles the communication with the server, i.e. send user commands and interprets the data sent by the server.

User agent type and version affect how the server should format the data sent to the End-user.

User interface development
The part of Development work that produces things needed by the Service to generate the user interface, e.g. an HTML page. This includes e.g. template development.

User pull Service type where a mobile terminal requests data from an application.

User push Service type where a mobile terminal sends data to an application.

View View defines what information is shown of the Game session and how it is formatted. Usually only part of the state information is included at a time. The view may be selected by the game or by the user of the game.

Wireless Without wires. Communication without any physical connections between the sender and the receiver. Using the radio frequency spectrum (airways) and hardware, software and technologies to transmit information.

Wireless terminal Any mobile terminal, mobile station, personal station, or personal terminal using non-fixed access to the network

3.3 Abbreviations

API API, or Application Programming Interface, is the interface by which an application program accesses operating system, platforms or other system services. An API provides a level of abstraction between the requesting application and the provider of the service, ensuring portability of the code. API main task is the translation of parameter lists from the calling format to the service provider format, while supporting the interpretation of call-by-value and call-by-reference arguments in one or both directions.

EDGE Enhanced Data rates for GSM Evolution.

EMS Enhanced Messaging Service.

GEE Game Execution Environment

GPRS General Packet Radio Services.

GSM Global System for Mobile Communication.

MMS The Multimedia Message Service (MMS) is a standard which will offer users the ability to send and receive messages consisting of text, sounds and video.

MO Mobile originated.

MT Mobile terminated.

PDC Personal Digital Cellular

SMS The Short Messaging Service (SMS) - originally part of the GSM system, it refers to any text messaging service available on digital mobile phones.

TDMA	Time Division Multiple Access.
UMTS	Universal Mobile Telecommunications System.
WAP	The Wireless Application Protocol (WAP) is an open, global specification that empowers mobile users with wireless devices to easily access and interact with information and services instantly. WAP is used with handheld digital wireless devices such as mobile phones, pagers, two-way radios, smartphones and communicators -- from low-end to high-end.
WCDMA	Wideband Code-Division Multiple Access
WLAN	Wireless Local Area Network.
XHTML	Extensible Hypertext Markup Language.

4. Introduction

This document specifies the requirements for an OMA gaming platform.

The primary audience of this document are developers of mobile gaming platforms. However, game developers will also find this document useful in determining the scope of functionality for an OMA gaming platform that is addressed by this release.

This specification address the issues of portability and interoperability in the mobile games space. This specification will allow game developers to produce and deploy mobile games that can be more easily ported between multiple gaming platforms and wireless networks, and played over different mobile devices.

This document is intended for readers familiar with the Java programming language.

In order to achieve compliance with this specification, the gaming platform supporting each of the APIs listed within this document is mandatory.

5. Session Management

5.1 Description

The Session Management APIs describe a framework and high level structure for the applications executing within this gaming platform. This framework controls the application's lifecycle. It also facilitates managing the Actors, meaning the End-User representation within the specific Application, and provides the Application Developer with access to all other interfaces and APIs necessary to create the application. These include, the Connectivity, Logging, Scoring, etc.

APIs are event based. Each event handling is a separate transaction. There are numerous types of events: user input events, timer events, etc. The full list of possible events is found below. To connect between the game logic and the gaming platform event handling mechanism the game logic source classes **must** implement one or more of supplied event listener interfaces. Those game logic classes should be registered in the gaming platform by the deployment process.

Game board lifecycle in the gaming platform is based on session entities. A session is defined as a series of interactions encompassing the Actor's lifetime within the specific Application Instance. `ActorSession` represents the single player role in a game board. In the real world, games may contain several players, and one single person can play several roles in different games or events in a single game board (person plays chess with himself), therefore the relation between `ActorSession` and `User` (which represents a user) is many to one. The relation between `ActorSession` and `ApplicationInstance` (which represents a single game board) may similarly be many to one. The `Application` session entity is used to represent a single type of game, registered in the gaming platform, and defines shared functionality between all `ApplicationInstances` of the same kind. `Actor` is shared functionality between all `ActorSessions` of same `User` in the same `Application`. More information about functionally provided in each session interface is provided in the detailed documentation in JavaDoc Appendix A.

The gaming platform **must** maintain persistent relations among all session objects for all existing game boards. Session objects and the relations among them may be changed automatically by a gaming platform as a result of some event, e.g. user input can create new `ActorSession`, or as a result of game logic execution, e.g. as result of `ActorSession.delete()` call.

When an event arrives at the system, the transaction opens and a transaction context is created. Among other things context contains the target of the handling event. There are two kinds of event, the first targets `ApplicationInstance` session entity, while the second targets `ActorSession` session entity. The gaming platform **must** create the correct context for every arriving event.

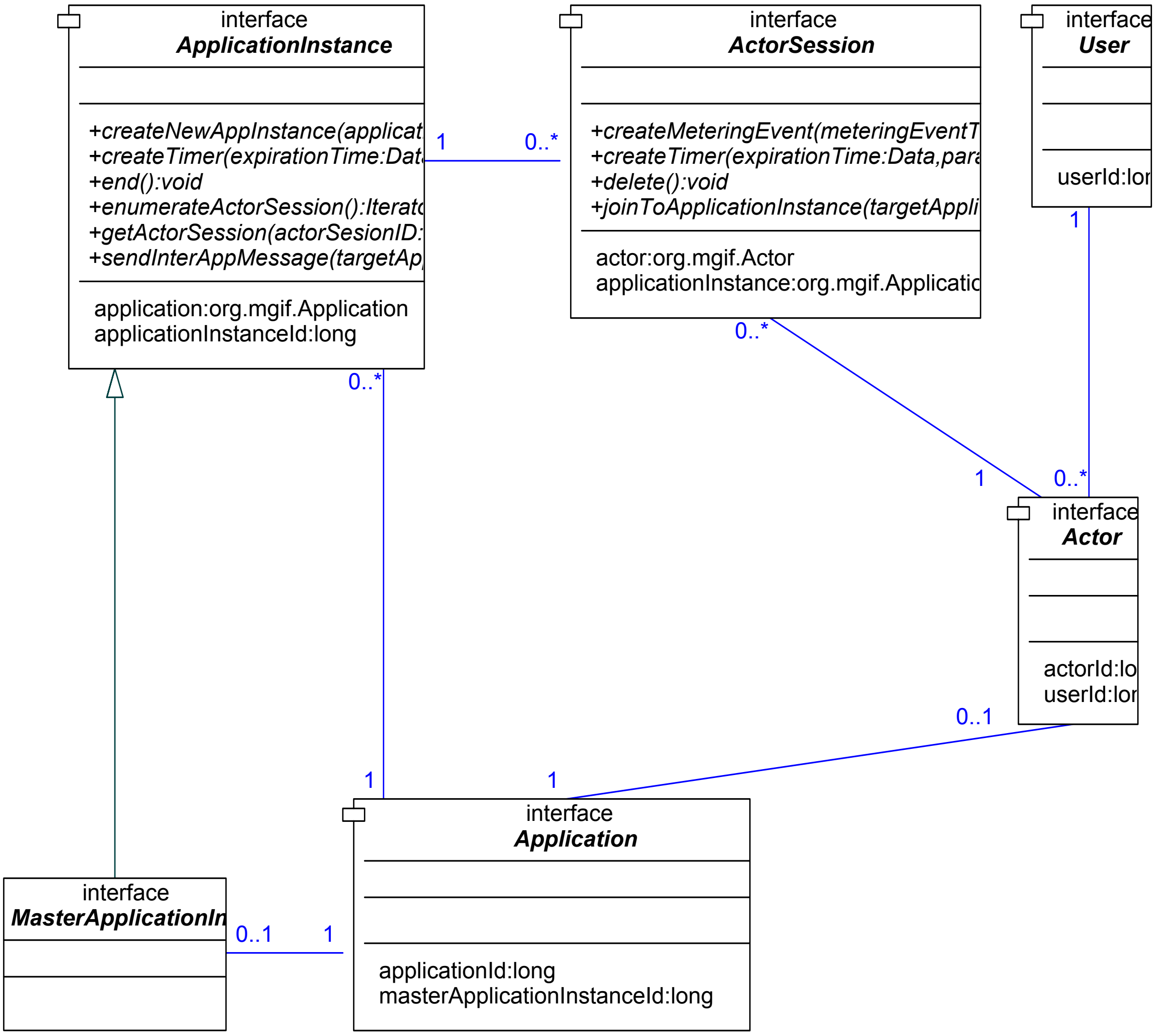

interface
ApplicationInstance
+createNewAppInstance(applicat
+createTimer(expirationTime:Dat
+end():void
+enumerateActorSession():Iterat
+getActorSession(actorSesionID:
+sendInterAppMessage(targetAp
application:org.mgif.Application
applicationInstanceId:long
interface
ActorSession
+createMeteringEvent(meteringEventT
+createTimer(expirationTime:Data,para
+delete():void
+joinToApplicationInstance(targetAppli
actor:org.mgif.Actor
applicationInstance:org.mgif.Applicatio
interface
User
userId:lor
interface
Actor
actorId:lo
userId:lor
interface
Application
applicationId:long
masterApplicationInstanceId:long
interface
MasterApplicationIn
1
0..*
1
0..*
0..*
1
0..*
1
1
0..1
0..*
1
0..1
1

Figure 1. Relations between main session entities

5.1.1 Interfaces

5.1.1.1 Actor

Actor represents shared information between all ActorSessions in same Application. It contains only an ActorID that is used to access this shared information. This state is created when a user starts to use Application for first time, and exists forever.

5.1.1.2 ActorSession

An ActorSession object represents a specific user in the context of a particular ApplicationInstance. Each user can be present in multiple applications at the same time, and thus be associated with several ActorSession objects. There is a one-to-many relationship between the Actor and ActorSession.

An ActorSession is created by the gaming platform and joined to the appropriate ApplicationInstance. ActorSession interface contains ActorSessionID which combined with ApplicationInstanceID and ApplicationID can be used to access ActorSession persistent state.

The corresponding ActorSession object in the game handles inputs received from a specific user.

An ActorSession is the actual representation of a user session in a game.

5.1.1.3 Application

An Application is the installed code or logic of a game. The Application is used for creating specific running instances: game boards. Application defines shared information for all ApplicationInstances. It also contains ApplicationID that is used to manage this information.

5.1.1.4 ApplicationInstance

The ApplicationInstance is the actual game played. It is the running instance of the Application object. For a game to be created there must be a new ApplicationInstance created to manage the actual game and the ActorSessions in it. In this way, multiple instances of an application can be run simultaneously, each controlling a different game and its users. A specific Tic-Tac-Toe board is an example of an ApplicationInstance.

Every ApplicationInstance contains an ApplicationInstanceID which may be used to access the persistent information.

ApplicationInstance is target for Application events and implements listeners for those events.

An ActorSession is the actual representation of a user session in a game.

5.1.1.5 ApplicationInstance - ActorSession Relations

Optional for this version of the OMA gaming platform:

An ApplicationInstance may manage several actors simultaneously.

A user may be represented simultaneously in several ApplicationInstances. A user may simultaneously play in all those game boards.

Note: The decision as to whether a user may play simultaneously in several instances of the same game is a commercial production decision of the operator. The application cannot make any assumptions to that effect.

5.1.1.6 MasterApplicationInstance

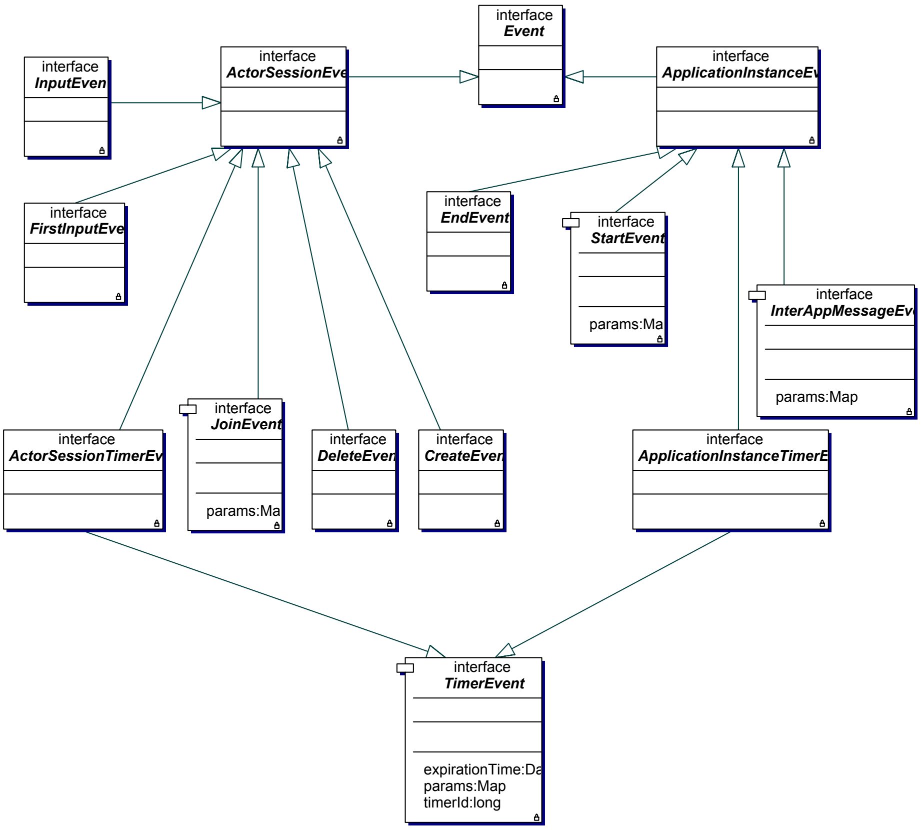
`MasterApplicationInstance` defines one special application instance, which is used to manage events and information shared for regular application instances. This instance is the target for special management events, e.g. declarative timers.

5.1.1.7 User

Users are subscribers who have cellular accounts and may access the gaming platform. The term "user" represents a specific person connecting to the system via a cellular phone or another communication device. The user is an object that exists independently of any game. The user places requests to the system to play a particular game. Normally a user is identified with a SIM (Subscriber Identity Module) or User Name and Password. Each user has an internal unique `UserID` on the gaming platform.

When the user is connected to an `Application`, an instance of the `ActorSession` class is created, by the gaming platform, for the user, and joined to that `Application` in the form of an `ActorSession` object. It is possible for multiple `ActorSession` objects to exist simultaneously for a particular user when each `ActorSession` object is attached to an `ApplicationInstance` but controlled by the same user.

5.2 Events List



5.2.1 Events

5.2.1.1 Actor Session

Base interface for all events that have actor as a target

5.2.1.2 Actor Session Timer

Indicates timer event for Actor object

5.2.1.3 Application Instance

Base interface for all events that have application instance object as target.

5.2.1.4 Application Instance Timer

Indicates timer event for Application Instance object

5.2.1.5 Create

Indicates start of lifecycle of Actor Session object.

5.2.1.6 Delete

indicates end of lifecycle of Actor Session object.

5.2.1.7 Delivery

Base interface for delivery reports.

5.2.1.8 ActorSessionDelivery

Event created as a result of delivery report to ActorSession object.

5.2.1.9 ApplicationInstanceDelivery

Event created as a result of delivery report to ApplicationInstance object.

5.2.1.10 End

Indicates end of lifecycle of Application Instance object.

5.2.1.11 Event

Base interface for all possible events

5.2.1.12 Input

Base class for all input events.

5.2.1.13 AsyncInputEvent

Event created as a result of input to ActorSession object

5.2.1.14 SyncInputEvent

Event created as a result of request/response input to ActorSession object

5.2.1.15 Inter App Message

Indictes message sent to Application Instance object by some other Application Instance object.

5.2.1.16 Join

Indicates request for join of user from some Actor Session object to another freshly created Actor Session object. This event immediately follows CreateEvent.

5.2.1.17 Start

5.2.1.18 Timer

Base class for al timer events.

5.3 Action Listeners List

5.3.1.1 OnActorSessionDelivery

This interface declares that implementing ActorSession class is ready to listen for incoming delivery reports.

5.3.1.2 OnActorSessionJoin

This interface declares that implementing ActorSession class is ready to listen for JoinEvent from some other ApplicationInstance and implements a hook to deal with transferred event.

5.3.1.3 OnActorSessionTimer

This interface declares that implementing ActorSession class is ready to listen for TimerEvents implements a hook to deal with transferred event. To create TimerEvent use ActorSession.createTimer() call.

5.3.1.4 OnApplicationInstanceDelivery

This interface declares that implementing ApplicationInstance class is ready to listen for incoming delivery reports.

5.3.1.5 OnApplicationInstanceTimer

This interface declares that implementing ApplicationInstance class is ready to listen for TimerEvents implements a hook to deal with transferred event. To create TimerEvent use ApplicationInstance.createTimer() call.

5.3.1.6 OnAsyncInput

This interface declares that implementing ActorSession class is ready to listen for asynchronous input event and implements a hook to deal with transferred event.

5.3.1.7 OnCreate

This interface declares that implementing ActorSession class is ready to listen for CreateEvent and implements a hook to deal with transferred event. This event is automatically created when ActorSession created, and this is first event that should be handled by ActorSession.

5.3.1.8 OnDelete

This interface declares that implementing ActorSession class is ready to listen for CreateEvent and implements a hook to deal with transferred event. This event is automatically created when ActorSession created, and this is first event that should be handled by ActorSession.

5.3.1.9 OnEnd

This interface declares that implementing ApplicationInstance class is ready to listen for EndEvent and implements a hook to deal with transferred event. This event is automatically created when ApplicationInstance deleted, and this is last event that should be handled by ApplicationInstance.

5.3.1.10 OnFirstAsyncInput

This interface declares that implementing ActorSession class is ready to listen for first asynchronous input event and implements a hook to deal with transferred event.

5.3.1.11 OnFirstSyncInput

This interface declares that implementing ActorSession class is ready to listen for first synchronous input event and implements a hook to deal with transferred event.

5.3.1.12 OnInterAppMessage

This interface declares that implementing ApplicationInstance class is ready to listen for InterAppMessageEvent and implements a hook to deal with transferred event.

5.3.1.13 OnStart

This interface declares that implementing ApplicationInstance class is ready to listen for StartEvent and implements a hook to deal with transferred event. This event is automatically created when ApplicationInstance created, and this is first event that should be handled by ApplicationInstance.

5.3.1.14 OnSyncInput

This interface declares that implementing ActorSession class is ready to listen for synchronous input event and implements a hook to deal with transferred event.

6. Connectivity

6.1 Description

The purpose of the connectivity APIs is to enable communication between the application and the clients. The connectivity APIs specify how the requests from clients are exposed to the applications, and how applications generate responses to the clients.

The communication models required by different application types can be categorized into four modes:

- client pull
- client push
- application pull
- application push

This version of the connectivity APIs only addresses messaging and browser clients. Subsequent versions will include executable clients.

6.2 Content

This section illustrates the components that collectively comprise the Connectivity APIs. This API comprises of three parts:

- synchronous communication
- asynchronous communication
- transfer, dealing with functionality common to each of the above types of communications

The Session APIs provide listener hooks for both synchronous and asynchronous communication. An application serving requests from both communication types can simply provide implementation for the both `onSynchInput`() and `onAsynchInput`() methods. The protocol used by the client determines which method the gaming platform calls whenever a new request is received from a client. The routing of requests to the correct application instance is implementation specific.

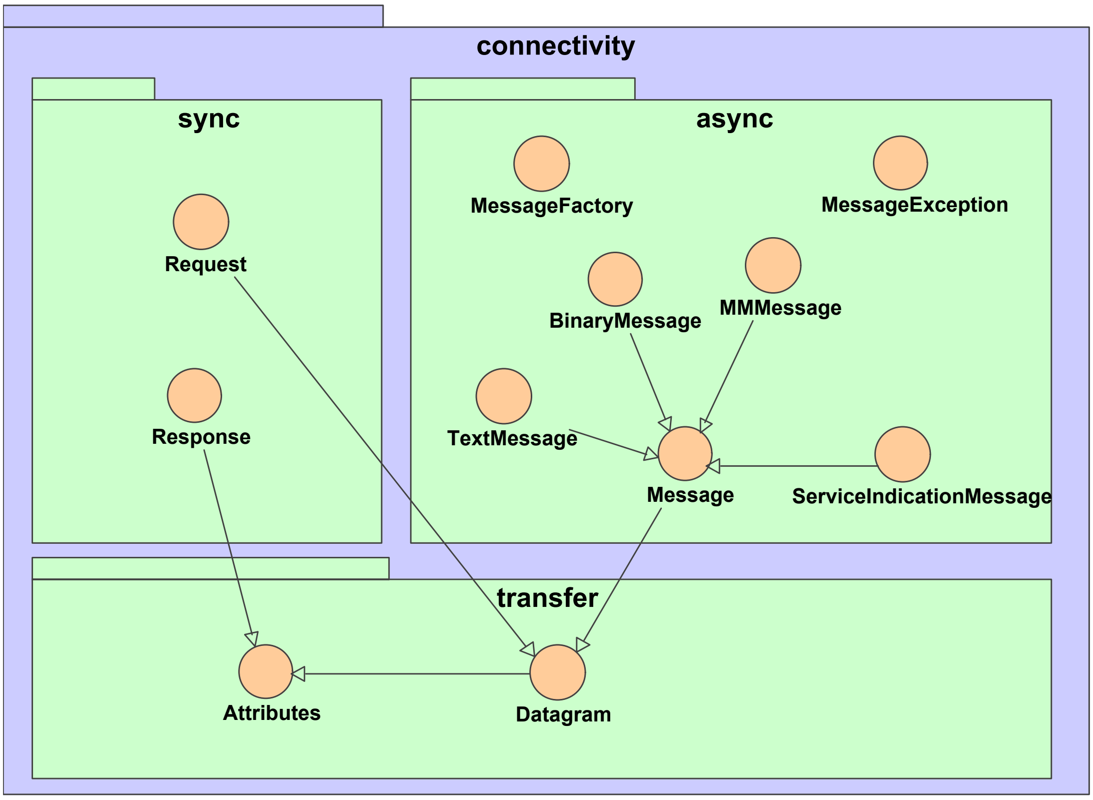


Figure 2. Connectivity package

6.3 Async Package

The `async` package manages the asynchronous communication.

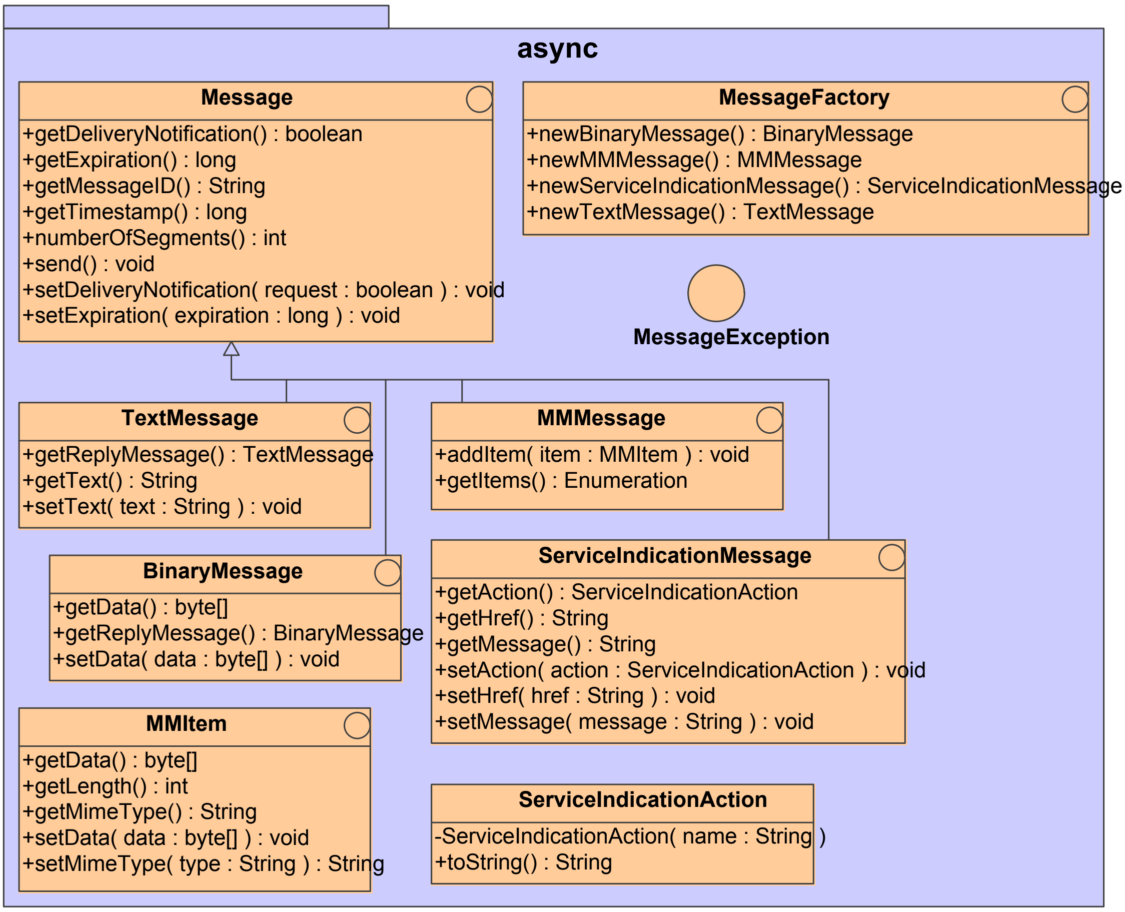

async
Message
+getDeliveryNotification() : boolean
+getExpiration() : long
+getMessageID() : String
+getTimestamp() : long
+numberOfSegments() : int
+send() : void
+setDeliveryNotification(request : boolean) : void
+setExpiration(expiration : long) : void
MessageFactory
+newBinaryMessage() : BinaryMessage
+newMMMessage() : MMMessage
+newServiceIndicationMessage() : ServiceIndicationMessage
+newTextMessage() : TextMessage
MessageException
TextMessage
+getReplyMessage() : TextMessage
+getText() : String
+setText(text : String) : void
MMMessage
+addItem(item : MMItem) : void
+getItems() : Enumeration
BinaryMessage
+getData() : byte[]
+getReplyMessage() : BinaryMessage
+setData(data : byte[]) : void
ServiceIndicationMessage
+getAction() : ServiceIndicationAction
+getHref() : String
+getMessage() : String
+setAction(action : ServiceIndicationAction) : void
+setHref(href : String) : void
+setMessage(message : String) : void
MMItem
+getData() : byte[]
+getLength() : int
+getMimeType() : String
+setData(data : byte[]) : void
+setMimeType(type : String) : String
ServiceIndicationAction
-ServiceIndicationAction(name : String)
+toString() : String

Figure 3. Async package

The asynchronous communication uses messages. The base interface for all messages is `org.mgif.connectivity.async.Message`. The interface provides basic common methods for handling messages.

6.3.1 Mobile originated messages

Incoming messages are delivered to the application via the `OnFirstAsyncInput` and `OnAsyncInput` listeners.

6.3.2 Mobile terminated messages

Message interfaces contain a `getReplyMessage()` method that provides an easy way to generate a response to an MO message. The `MessageFactory` is used for pushing an MT message to a user. The `MessageFactory` is obtained from the `ActorSession`.

Sample code snippet in Java (receiving and responding to an asynchronous client request):

```
import org.mgif.connectivity.async.*;

...
 public void onAsyncInput(AsyncInputEvent event)
 {
   TextMessage textMessage = (TextMessage) event.getMessage();
   if (textMessage.getText().equals("Hello GAMERS!"))
   {
     TextMessage replyMessage
       = textMessage.getReplyMessage();
     replyMessage.setText("Hello!");
     replyMessage.send();
   }
 }
```

The basic message interface is inherited to produce service specific message types.

- `org.mgif.connectivity.async.TextMessage` is an interface for handling messages containing only textual data.
- `org.mgif.connectivity.async.BinaryMessage` is an interface for handling messages containing binary data, e.g. operator logos.
- `org.mgif.connectivity.async.ServiceIndicationMessage` is an interface for handling WAP push.
- `org.mgif.connectivity.async.MMMessage` is an interface for handling Multimedia messages

For details please refer to the Javadoc of the corresponding interfaces.

6.3.3 Sync package

The sync package handles the synchronous communication.

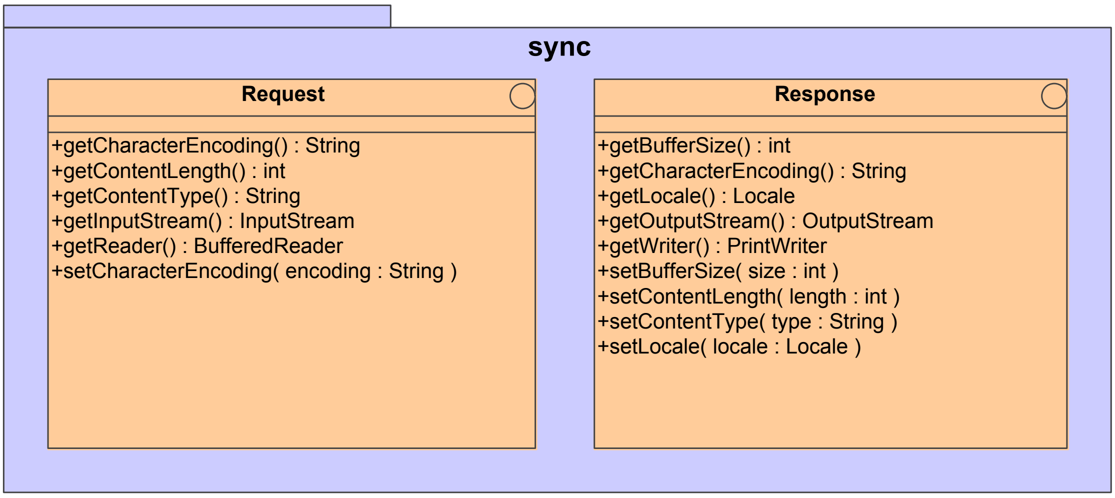

sync
Request
+getCharacterEncoding() : String
+getContentLength() : int
+getContentType() : String
+getInputStream() : InputStream
+getReader() : BufferedReader
+setCharacterEncoding(encoding : String)
Response
+getBufferSize() : int
+getCharacterEncoding() : String
+getLocale() : Locale
+getOutputStream() : OutputStream
+getWriter() : PrintWriter
+setBufferSize(size : int)
+setContentLength(length : int)
+setContentType(type : String)
+setLocale(locale : Locale)

Figure 4. Sync package

Synchronous communication uses the request/reply paradigm. This is realized by using the `org.mgif.connectivity.sync.Request` and `org.mgif.connectivity.sync.Response` interfaces. Both interfaces contain methods for getting and setting attributes, as well as other methods for dealing with the actual content of the request or response.

Sample code snippet in Java (receiving and responding to an synchronous client request):

```
import org.mgif.connectivity.sync.*;

...

 public void onSyncInput(SyncInputEvent event)
 {
   Request request = event.getRequest();
   if (request.getContentType().equals("text/html"))
   {
     Response response = event.getResponse();
     response.getWriter().println("<html>");
     response.getWriter().println("<h1>Hello</h1>");
     response.getWriter().println("</html>");
   }
 }
```

6.3.4 Transfer package

The transfer package contains the interfaces extended by both the async and sync packages.

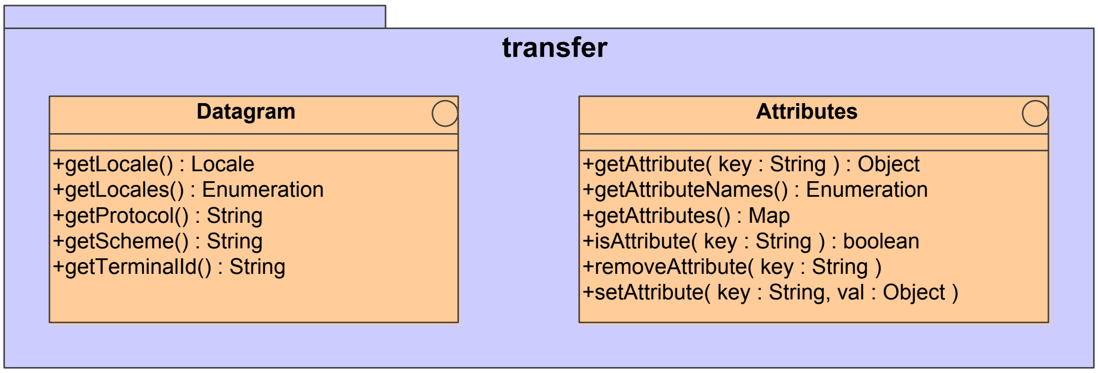

transfer
Datagram
+getLocale() : Locale
+getLocales() : Enumeration
+getProtocol() : String
+getScheme() : String
+getTerminalId() : String
Attributes
+getAttribute(key : String) : Object
+getAttributeNames() : Enumeration
+getAttributes() : Map
+isAttribute(key : String) : boolean
+removeAttribute(key : String)
+setAttribute(key : String, val : Object)

Figure 5. Transfer package

7. Metering

7.1 Description

A gaming platform **may** produce metering events for potential use in different billing scenarios. Some metering events are traffic related, e.g. game session duration, MT SMS message, and some are application specific, e.g. moving to the next level in a game.

A gaming platform **may** use metering events for post-paid, pre-paid and subscription scenarios.

The scope of these APIs is to address the generation of game specific metering events.

7.2 Traffic Based Events

A gaming platform **may** implement advanced, flexible metering services. A gaming platform **may** provide metering of:

- Session duration
- Data transfer
- MT and MO message
- Subscription/Pre-Paid Billing authorization

The application developer is not responsible for adding code for metering of traffic related events. Such metering is handled transparently by the gaming platform.

7.3 Game Specific Events

The Metering APIs are used to programmatically commit metering events. Each metering event is created using a metering event type `ID`. All metering event types **must** be configured prior to use. Processes or tools for configuration are undefined. A gaming platform vendor may choose the mechanism for configuring these events. Each metering event type **must** have an integer `ID`. Metering events are created in the scope of an `ActorSession`.

Sample code snippet in Java (metering event creation):

```
import org.mgif.*;
import org.mgif.metering.*;

...

// Create a metering event for moving to the next level
// (ID=17)
// An event type with ID=17 must have been configured
```

```
MeteringEvent event = ActorSession.createMeteringEvent(17);
event.raise();
```

8. Score and Competition Management

8.1 Description

These APIs provide the mechanisms for recording and retrieving scores. This allows various forms of competition to be provided by either the gaming platform or the individual game application.

8.1.1 Scoring

The gaming platform **must** support the following scoring models for applications. An application **must** state the scoring model it will use for each of the scores it records, and **must** reliably record these scores for each player for each session following the stated model.

Each model may be further configured according to scoring information, e.g. May more than one score be stored for a particular user? Or are scores “better” when higher or lower?

8.1.1.1 No Scoring

No scores are recorded and no score table information is available. No further configuration required. This is often used in games in which there is no simple score available. Such games need to implement their own internal competition mechanisms as they will not be able to assume support from the gaming platform.

8.1.1.2 Simple Numeric Scores

The game service records a simple numeric score for each user. This model **must** be configured according to scoring information:

- Is “better” higher or lower?
- The number of scores that may be recorded per user.
- The total number of scores that are to be recorded.

The game application **must** record a score at least once for each player in each game session.

8.1.1.3 Cumulative Numeric Scores

The user builds up a score over a number of sessions. This scoring model **must** be configured according to scoring information:

- Is “better” higher or lower?
- The total number of scores that are to be recorded.

In this case the game application first retrieves the user’s current score, alters it based on the outcome of the session and then records the updated score.

8.1.1.4 Rank

This scoring model is provided to support services that maintain their own score table internally by some mechanism not supported within this specification. The game service simply calls setScore with the players new rank as the value – all lower ranked positions are moved down to make room. This scoring model **must** be configured according to scoring information:

- The total number of scores that are to be recorded.

8.1.1.5 Combined

An application can specify any number of score tables, each using one of the model described above.

8.1.2 Competitions

Competition management is normally a feature of the game server and not the individual game, consequently there is no specific API provided to support competitions. For a games service to be useable for competitions it **must** record a score for every user in every game session. Such scores **must** always be reliably comparable.

For every such games service the game platform **may** provide facilities for competitions running over various time periods with the winner selected in a variety of ways – in part depending on the nature of the scoring used in the game service in question.

If a game service wishes to implement a service specific competition system it may do so using a combination of the Score Management API and Scheduling API. This is discouraged as it is likely to duplicate platform functionality.

8.2 Content

8.2.1 Scores

All scores returned from the various interfaces defined below return objects implementing the `org.mgif.score.Score` interface. This is a simple bean style interface allowing access to the score value, the rank it represents from where the score was retrieved, if applicable, when the score was achieved and who achieved it.

Score values are always represented by the `int` type. If a game requires fractional scores it should scale these to produce an integer representation with an appropriate number of significant digits.

8.2.2 Multiple Score Tables

An application can specify any number of score tables within reason. These are referred to via a table number. All of the APIs in this chapter are provided in two forms. The first form does not specify a table number and operates on the first or only table. The second form allows the table number to be specified.

8.2.3 Recording Scores

All score recording is done via an object implementing the `org.mgif.score.ScoreManager` interface that can be retrieved from the `ActorSession`.

The setScore method allows a score value to be set for the session. Normally it is assumed the score was achieved at the time the setScore() method was called. Optionally the time/date it was achieved may be provided explicitly.

The getScore() method retrieves the last score set for this session, or from a previous session if the cumulative model is in use.

8.2.4 Retrieving Past Scores

All score retrieval is done via an object implementing the org.mgif.score.ScoreTableManager interface which can be retrieved from the ApplicationInstance.

The simplest method is getScoreAt() which retrieves a single score from a particular rank. This will return null if no score is available at that rank.

The other methods all return an array of Score objects that may vary in size between 0 and the requested number of scores if the scores requested are available or are not in the table in question.

The top scores in the table can be retrieved by using the getTopScores() method.

The scores around a particular rank, or the rank of the highest score of some specific Actor, may be retrieved by using the getScoresAround() method.

9. Timers

9.1 Description

Applications that need to delay or schedule activities for a later time should use the timer service provided by the gaming platform. The `Timer` service provides scheduling and notification of timers. Agaming platform **may** provide additional unspecified services, e.g. load balancing and persistence of timers.

Timers can be created either programmatically, i.e. by calling a method on an interface, or declaratively, i.e. via some implementation specific mechanism for configuring timers.

9.2 Programmatic Timers

There are two types of programmatic timers, `ActorSession` timers and `ApplicationInstance` timers.

`ActorSession` timers are created and notified in the scope of an `ActorSession.` The `createTimer()` method on the `ActorSession` interface is used for creating these timers. When such a timer expires the method `onActorSessionTimer()` on the `OnActorSessionTimer` interface is invoked.

`ApplicationInstance` timers are created and notified in the scope of an `ApplicationInstance.` The `createTimer()` method on the `ApplicationInstance` interface is used for creating these timers. When such a timer expires the method `onApplicationInstanceTimer()` on the `OnApplicationInstanceTimer` interface is invoked.

Sample code snippet in Java (create timer)

```
import org.mgif.*;

...
// Create a timer in ActorSession scope
Hashtable params = new Hashtable();
params.put("myParam", "myValue");
myActorSession.createTimer(new Date((new Date()).getTime() +
60*60*1000), params);

...
```

Sample code snippet in Java (timer notification)

```
import org.mgif.*;
import org.mgif.listener*;

public class MyTask implements OnActorSessionTimer {
  public void onActorSessionTimer(ActorSessionTimer event) {
    // This method is called when the timer expires
    Map params = event.getParams();
  }
}
```

9.3 Declarative Timers

Declarative timers are created via an implementation specific mechanism, e.g. a configuration file or administration tool. . When such a timer expires the method `onApplicationInstanceTimer()` on the `OnApplicationInstanceTimer` interface is invoked. All declarative timers are notified for the `MasterApplicationInstance` entity.

10.Logging

10.1 Description

A gaming platform **may** provide logging of events of all client requests and MT SMS messages, timer events, etc. For debugging, troubleshooting, monitoring and other purposes it can be important for the application developer to be able to write information to an application log. The logging APIs of the gaming platform specification is designed for these purposes.

10.1.1 Logger Interface

The `Logger` interface from logging APIs, can be used to add information to the log. The `ActorSession` and `ApplicationInstance` interfaces can be used to retrieve a `Logger`.

Sample code snippet in Java

```
import org.mgif.util.logging.*;

...

Logger logger = myActorSession.getLogger();
logger.fine("Moving to the next level.");
// Move to the next level
...
if (successful) {
   Log.info("The move to the next level was successful.");
} else {
   Log.warning("The move to the next level was
        unsuccessful.");
```

-

JavaDocs

The related JavaDocs [OMAGPJD10] for this Gaming Platform Specification are freely available in the OMA-GamingPlatform-JavaDocs-V1_0-20030525-D zipfile, which can be found at http://www.openmobilealliance.org/documents.asp.

Sample Code

```
package org.mgif.examples.RPS;

import org.mgif.listener.OnFirstAsyncInput;
import org.mgif.listener.OnAsyncInput;
import org.mgif.listener.OnActorSessionTimer;
import org.mgif.listener.OnDelete;
import org.mgif.event.AsyncInputEvent;
import org.mgif.event.DeleteEvent;
import org.mgif.event.ActorSessionTimerEvent;
import org.mgif.ActorSession;
import org.mgif.util.logging.Logger;
import org.mgif.score.Score;
import org.mgif.score.ScoreManager;
import org.mgif.connectivity.async.MessageFactory;
import org.mgif.connectivity.async.TextMessage;
import org.mgif.connectivity.async.Message;

import java.util.Map;
import java.util.Date;
import java.util.HashMap;

/**
 * A very simple Rock-Paper-Scissors text message game written to this API
 * The game automatically ends after 24 hours.
 */
public class RockPaperScissors implements OnFirstAsyncInput, OnAsyncInput, OnActorSessionTimer,
OnDelete
{
    /**
     * Number of milliseconds on 24 hours!
     */
    private static final int ONE_DAY = 1000*60*60*24;

    /**
     * The three possible values to play!
     */
    private static final int ROCK = 0;
    private static final int PAPER = 1;
    private static final int SCISSORS = 2;

    /**
     * Handle a first input in a session.
     */
    public void onFirstAsyncInput(AsyncInputEvent event)
    {
        Message mo = event.getMessage();
        ActorSession actorSession = event.getActorSession();
        Logger logger = actorSession.getLogger();

        // Create an MT message to send back out.
        TextMessage mt;
        if (mo instanceof TextMessage) {
            logger.info("RPS: Session started for "+mo.getOriginator()+" by a text message.");
            mt = ((TextMessage)mo).getReplyMessage();
        } else {
            logger.info("RPS: Session started for "+mo.getOriginator()+" by a non-text message.");
            mt = actorSession.getMessageFactory().newTextMessage();
        }

        // Fill in the MT and send it.
        mt.setText("Welcome to rock paper scissors - take your pick of r,p, or s");
        mt.send();

        // Create the timer which will end the game.
        Date oneDayInTheFuture = new Date((new Date()).getTime() + ONE_DAY);
        Map args = new HashMap();
        actorSession.createTimer(oneDayInTheFuture, args);
    }

    /**
     * Handle all subsequent inputs in a session.
     */
    public void onAsyncInput(AsyncInputEvent event)
    {
        Message message = event.getMessage();
        ActorSession actorSession = event.getActorSession();

        if (message instanceof TextMessage) {

            TextMessage mo = (TextMessage)message;

            // Try to act on the content of the MO.
            Result result = null;
            if (mo.getText().equalsIgnoreCase("r")) {
                result = takeTurn(ROCK);
            } else if (mo.getText().equalsIgnoreCase("p")) {
                result = takeTurn(PAPER);
            } else if (mo.getText().equalsIgnoreCase("s")) {
                result = takeTurn(SCISSORS);
            }

            TextMessage mt = mo.getReplyMessage();
            if (result!=null) {
                // We have a result so send out an appropriate MT.

                if (result.isUserWin()) {
                    // If the user won then update their score.
                    ScoreManager scoreManager = actorSession.getScoreManager();
                    Score currentScore = scoreManager.getScore();
                    scoreManager.setScore(currentScore.getValue()+1);
                }

                mt.setText(result.getDescription());

            } else {
                // We didn't understand the MO so send an error as out MT.
                mt.setText("I don't understand. Please choose one of r, p or s!");
            }
            mt.send();

        } else {
            // If the MO wasn't a text message push out an error.
            TextMessage mt = actorSession.getMessageFactory().newTextMessage();
            mt.setText("Rock-Paper-Scissors can only respond to plain text messages - sorry!");
            mt.send();
        }
    }

    /**
     * Handle timer events.
     */
    public void onActorSessionTimer(ActorSessionTimerEvent event)
    {
        ActorSession actorSession = event.getActorSession();
        MessageFactory messageFactory = actorSession.getMessageFactory();

        // Let the user know the time has run out!
        TextMessage mt = messageFactory.newTextMessage();
        mt.setText("Your game time has run out!");
        mt.send();

        // Close the session.
        actorSession.delete();
    }

    /**
     * Handle session end.
     */
    public void onDelete(DeleteEvent event)
    {
        ActorSession actorSession = event.getActorSession();
        ScoreManager scoreManager = actorSession.getScoreManager();
        MessageFactory messageFactory = actorSession.getMessageFactory();
        Logger logger = actorSession.getLogger();

        // Send the user a summary of the session.
        TextMessage mt = messageFactory.newTextMessage();
        mt.setText("Your final score was "+scoreManager.getScore());
        mt.send();

        logger.info("RPS: Session finished for "+mt.getDestination());
    }

    /**
     * Table of results against user and cpu choices.
     */
    private static Result results[] = {
        new Result(false, "Rock draws with rock."),
        new Result(false, "Paper wraps rock - you lose."),
        new Result(true, "Rock blunts scissors - you win!"),
        new Result(true, "Paper wraps rock - you win!"),
        new Result(false, "Paper draws with paper."),
        new Result(false, "Scissors cut paper - you lose."),
        new Result(false, "Rock blunts scissors - you lose."),
        new Result(true, "Scissors cut paper - you win!"),
        new Result(false, "Scissors draws with scissors.")
    };

    /**
```

```
     * The actual game "logic" - table driven.
     */
    private Result takeTurn(int userChoice)
    {
        int cpuChoice = (int)(Math.random()*3);
        boolean userWin = false;
        String description = null;

        Result result = results[userChoice*3+cpuChoice];

        return result;
    }

    /**
     * Simple bean representing the result of a round.
     */
    private static class Result
    {
        private boolean userWin;
        private String description;

        public Result(boolean userWin, String description)
        {
            this.userWin = userWin;
            this.description = description;
        }

        public boolean isUserWin()
        {
            return userWin;
        }

        public String getDescription()
        {
            return description;
        }
    }
}
```

Static Conformance Requirements (Normative)

The notation used in this appendix is specified in [CREQ].

Item	Function	Reference	Status	Requirement
OMA-GamingPlatform-V1_0-20030525-D			M	
OMA-GamingPlatform-V1_0-20030525-D -			O	

Change History (Informative)

A.1 Approved Version History

Reference	Date	Description
n/a	n/a	No previous version within OMA

A.2 Draft/Candidate Version 1.0 History

Type of Change	Date	Section	Description
Draft Version OMA-GamingPlatform-V1_0-20030525-D	25-May-2003		Draft submitted to OMA TP aproval
Candidate Version OMA-ERELD-Games-Services-V1_0-20030612-C	12 June 2003		Status Changed to Candidate by TP TP ref# OMA-TP-2003-0267R1