Server Overview

BigWorld Technology 2.0. Released 2010.

Software designed and built in Australia by BigWorld.

Level 2, Wentworth Park Grandstand, Wattle St
Glebe NSW 2037, Australia
www.bigworldtech.com

Table of Contents

Chapter 1. Introduction

BigWorld Technology is a BigWorld's middleware for implementing Massively Multiplayer Online Games. This document gives an overview of the current implementation of BigWorld Technology.

Note
For details on BigWorld terminology, see the document Glossary of Terms.

Chapter 2. Rules of Thumb

This is a list of rules/ideas/philosophies that have been used in the design.

- **Scalability, reliability, efficiency**

  The general goal is to produce a scalable, reliable, and efficient system. This should be done while keeping as much simplicity and flexibility as possible.

- **Occam's Razor**

  The simplest design that satisfies all requirements should be considered the best, or in the words of Einstein, "Make things as simple as possible, but no simpler".

- **Improve the worst case**

  In general (mainly when it comes to the client experience), the worst case should be improved over the average case. For example, it is not beneficial to have a blinding fast and accurate situation when a client is not near a cell boundary if the experience is poor when he is near one.

- **Client/server bandwidth is valuable**

  The most important resource is the bandwidth between the client and server. After this, it is probably CPU, and then intra-server bandwidth.

- **Keep information together; Avoid two-way calls**

  Information (or data) that is often used together should be easily accessed together. For example, a large amount of the data processed together in the game is related to objects that are geometrically close. It makes sense then to use data partitioning based on locality.

  It is also expensive to have to request information from a separate server machine when it is necessary. This is true for a number of reasons including the extra hops and coordination required and (maybe even more importantly) the reduced likelihood of being able to batch requests together.

- **Avoid bottlenecks; Make the system distributed**

  The design should try to avoid single central points where things occur. This approach can cause performance bottlenecks and make the design non-scalable. It can also introduce a single point of failure, therefore raising fault tolerance issues.

- **Where possible, do communication in batches**

  There is a fairly high overhead in sending a single packet. That is, it is a lot more expensive to send ten individual packets than it is to send one packet that is ten times bigger.

Chapter 3. Concepts

This section explains general concepts and issues relevant to the design.

3.1. Location of an Object's Data

There are four main locations for the active data of an object:

1. Associated with the cell part of an entity.
2. Associated with the base part of an entity.
3. In the persistent world database.
4. On the client.

Data associated with an entity on a cell can be categorised as:

- **Internal data**

  Used and stored only on the cell in which it lives.

- **Server (or ghosted) data**

  Available to other entities on the server.

- **Client data**

  Available to (at least some) client machines.

3.2. Actions

From the client's perspective, there are conceptually four types of action:

1. **Server action**

   This is an unsolicited action that comes from the server, and is not generated by this client.

   For example, another avatar jumping.

2. **Local action**

   This is an action that only occurs locally on the client, and does not need to be communicated to the server or other clients.

   For example, special effects like particles bouncing or a flame slightly flaring.

3. **Undoable action**

   This is an action that the client takes immediately, under the assumption that it is correct, and then communicates to the server. The server has the ability to disallow the action and make the client rollback the action.

   For example, the client stepping forward.

4. **Server-confirmed action**

   This is an action that needs to receive confirmation from the server to be performed on the client.

For example, the player wanting to shake hands with another player. Another example may be hitting a player (and thinking that it is dead) but not showing this until the server has confirmed it (this has some similarity to the Server action, except that the origin of the action was from this client).

3.3. Latency

The game designer needs to hide the latency from the player using as many latency-hiding tricks as possible. There are two main sources of latency:

1. **Internet latency**

   All information that travels between clients goes via the server, and so must have two trips on the Internet.

2. **Server latency**

   This is the time between information being received by the server, processing it, and responding back to the client. In an MMOG environment with large numbers of players, bandwidth is a precious resource, so not all new information can be sent out immediately. In BigWorld, the Priority Queue manages this.

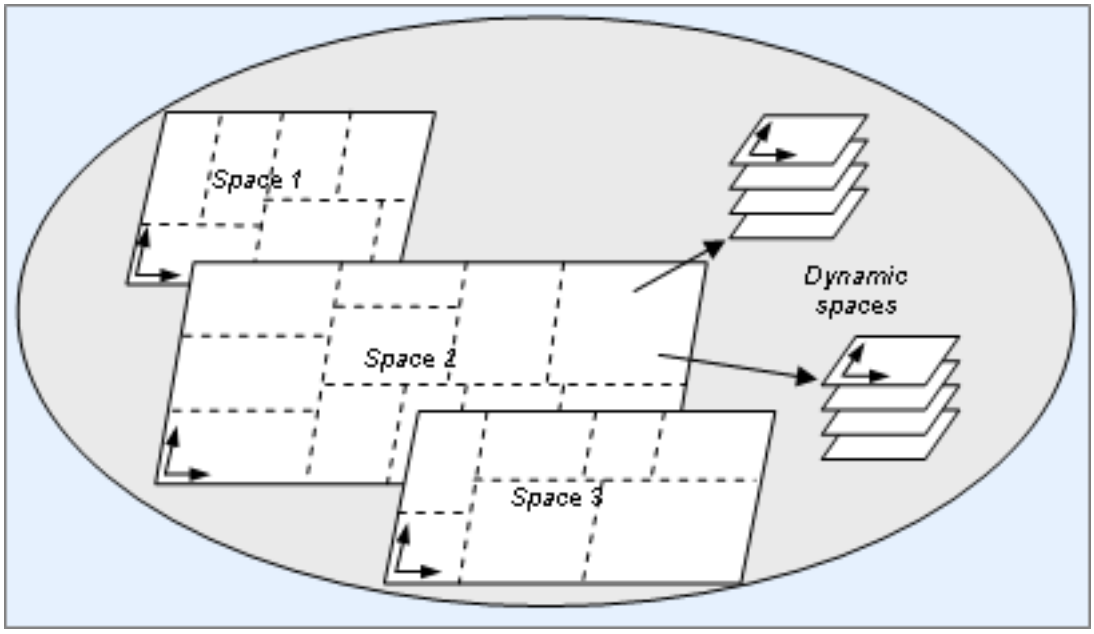
The Priority Queue can be tuned to reduce the latency on critical information caused by the server. This does not affect the Internet latency, though.

Latency can also be introduced on the client side. This can occur if the developer allows a delay between receiving data and sending it to the server, or between receiving data and displaying it.

3.4. Spaces and Cells

The game world is made up of multiple spaces. Each space is a continuous Euclidean region, spanned by a single coordinate system.

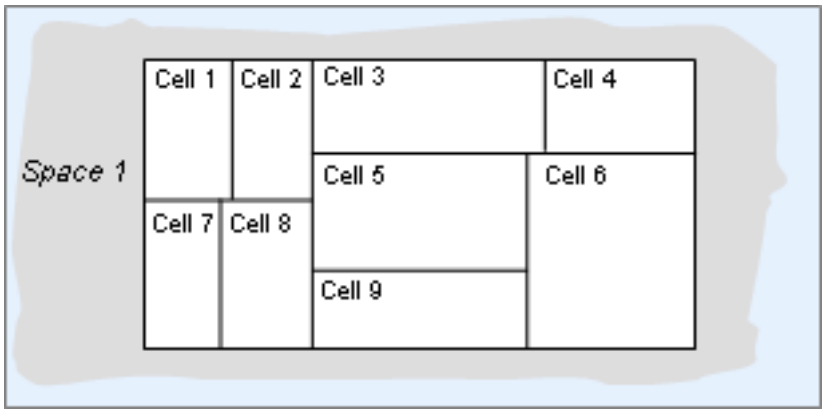
Spaces may be always present, or be dynamically created to enable a group of one or more players to adventure in isolation from other players. There may be many instances of a given space running the same geometry simultaneously and independently.



Visualisation of spaces and division in cells (cell boundaries marked in dotted lines)

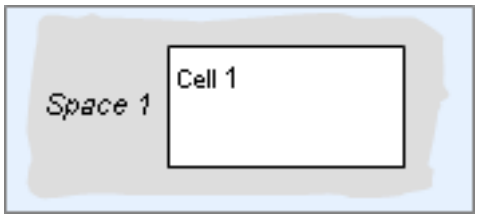
The space is a logical concept, dealt with in game script.

Cells exist at a more physical level. They divide large game spaces geometrically, for the purpose of load balancing across multiple CellApps.



A large space divided into cells

For a small space, a single cell is enough to cover it.



A small space covered by only one cell

3.5. Coordinate System

BigWorld uses a left-hand coordinate system. The x-axis points *"left"*, the y-axis points *"up"* and the z-axis points *"forward"*.

yaw is rotation around the y-axis. Positive is to the right, negative is to the left.

pitch is rotation around the x-axis. Positive is nose pointing down, negative is nose pointing up.

roll is rotation around the z-axis. Positive is to the left, negative is to the right.

Chapter 4. Design Introduction

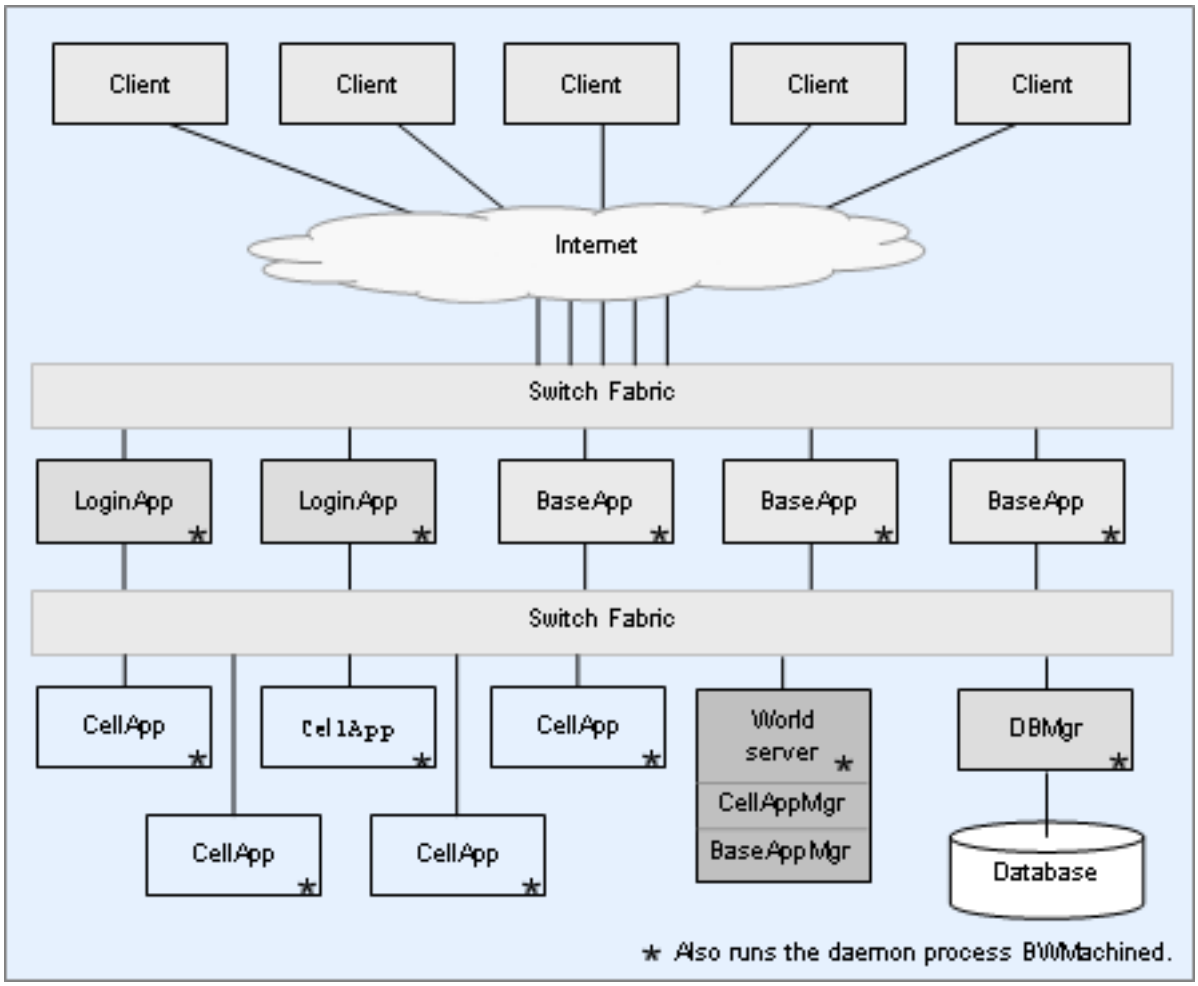
This section discusses the design of BigWorld Server environment, with a brief overview of its components and a series of Use Cases.

4.1. Hardware Components

The BaseApps and the LoginApps are the only server hardware that needs to be connected to the Internet.

For security purposes, it is recommended that the BaseApp and LoginApp machines have two network cards; one to connect to the Internet, and the other to connect to the rest of the server cluster.

The diagram below shows the connection between the different components of server hardware.



BigWorld Server components

4.2. Software Components

The primary software components of the server are:

- CellApp
- CellAppMgr

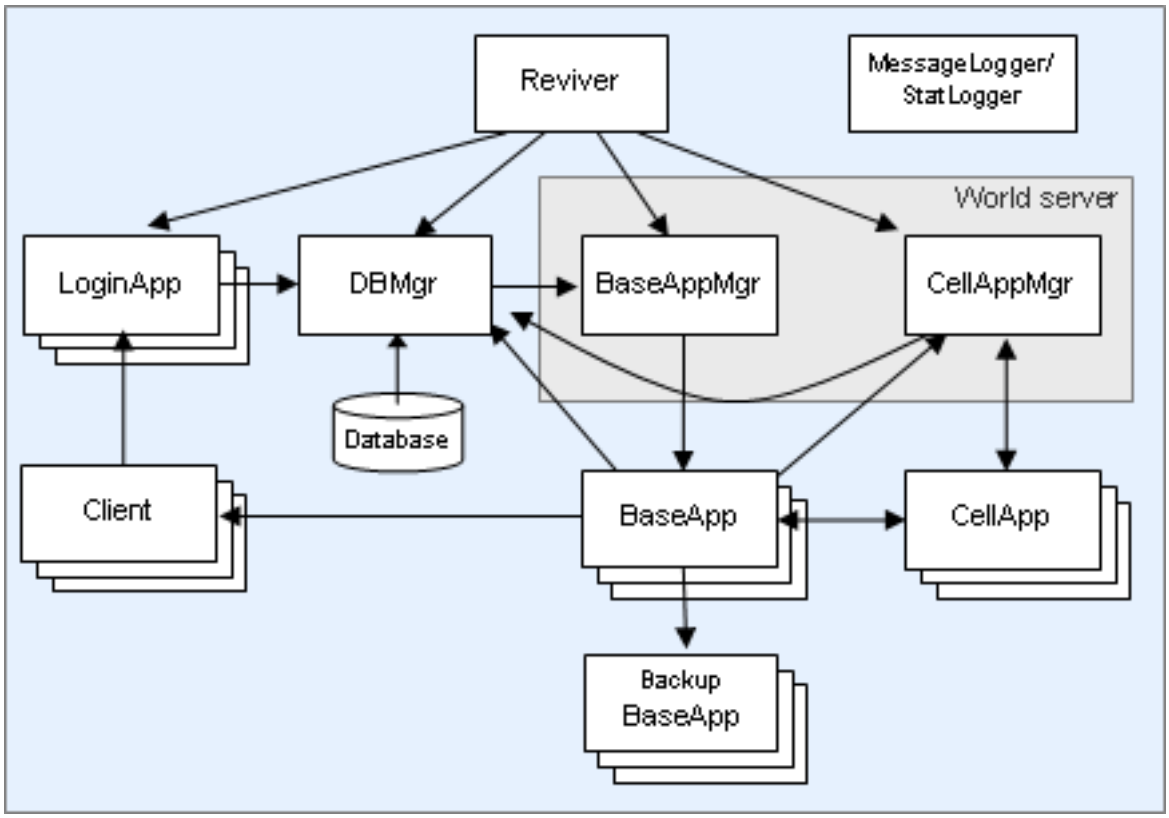
- BaseApp
- BaseAppMgr
- LoginApp
- DBMgr
- Reviver

There is also a special daemon process called BWMachined, which runs on each machine.

In a production environment, each CellApp runs on its own machine, as does each BaseApp. All other processes (apart from BWMachined) can be organised to run in various combinations across one or more servers, depending on system loads.

In the hardware diagram above, LoginApp runs on its own machine(s), directly connected to the Internet. The DBMgr also runs on its own machine, whereas the CellAppMgr and BaseAppMgr both run on the World server.

The diagram below shows the communication paths between many of the software components:



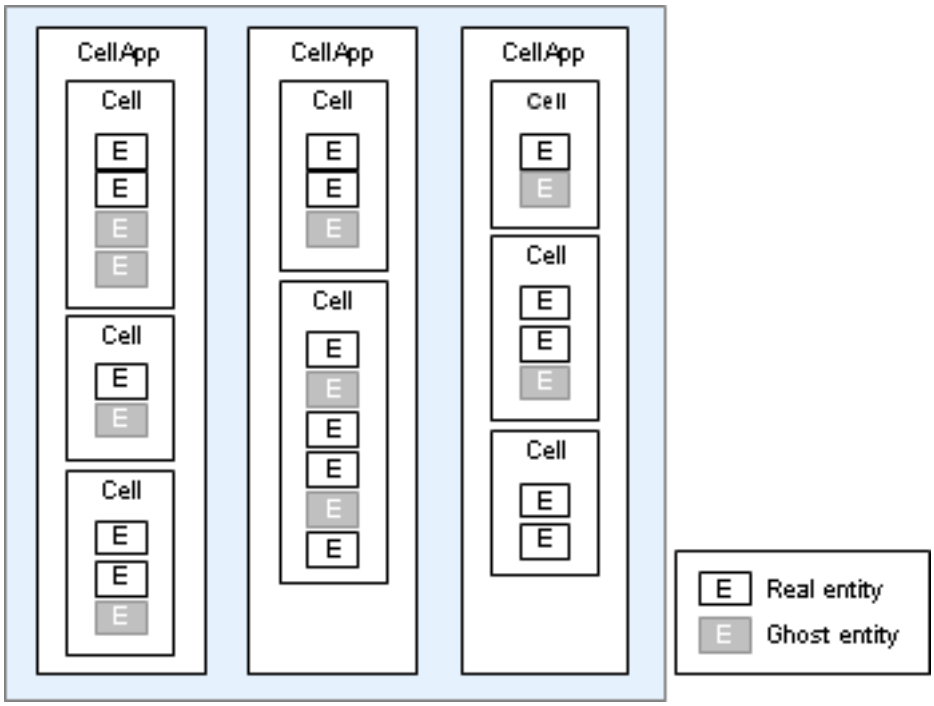
Communication between BigWorld components

4.2.1. CellApp

Each CellApp is responsible for all cells running on it.

CellApps are probably the most important part of the architecture. Each cell is responsible for either part of a large space or an entire one. Within the game world, cells do not overlap, and collectively they cover all game spaces.

Each cell is responsible for maintaining the entities located within its boundaries. The entity is the basic element of operations within the BigWorld game environment. Its distinguishing feature is its point position in a space. A cell may also keep ghosts (copies) of entities that are near, but outside its own boundary.

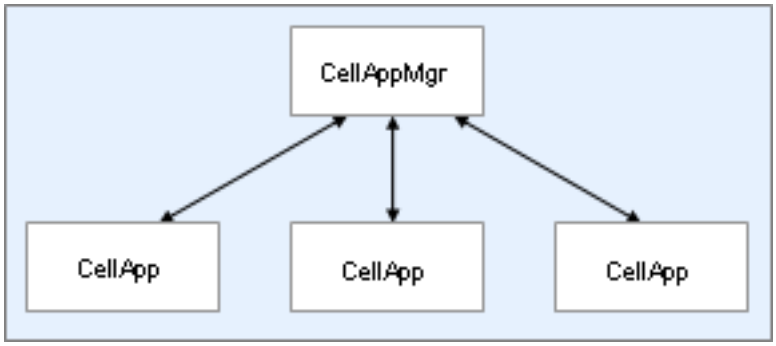


CellApps managing cells

4.2.2. CellAppMgr

The main responsibility of the CellAppMgr is to direct the cells and CellApps.

It coordinates which cells run on which CellApps, and balances the load on each of them by varying the size of the cells.



CellAppMgr managing CellApps

4.2.3. BaseApp

In some respects, BaseApps can be seen as the firewalls for the server.

For the client, their main purpose is to isolate it from transitions of its entity between CellApps.

Each BaseApp contains many bases.

Connected clients are served by an enhanced base known as a proxy. Each proxy is responsible for at most one client. Each client talks to one proxy. This proxy is responsible for redirecting messages from the client to the correct cell.

In general, the BaseApp maintains bases. A base represents an object or function that does not need to have a position in the world.

For example, an object used for group chat is a base without a corresponding cell entity.

Any cell entity can have a corresponding base entity, giving it a fixed point of contact in the game world.

A base is also considered to be part of an entity, so a proxy is the 'base entity' for a client, which will usually have a related 'cell entity' part, as well as a 'client entity' part instantiated on any clients that can see it.

4.2.4. BaseAppMgr

The game world has a single BaseAppMgr running, which is responsible for managing the BaseApps.

Its main job is to allocate new client connections to the most appropriate BaseApp, and keep track of them.

4.2.5. LoginApp

Clients talk to this component to initiate a session with the server. The LoginApp then adds the player as a proxy on a BaseApp, which may go on to create an entity on a CellApp.

Multiple instances of this component can be running at once.

4.2.6. DBMgr

DBMgr is the interface to the database, where the persistent state of the world is stored.

When a player logs in, the login process requests from the database the full set of properties for that player's entity. This data is used to instantiate a proxy for the player on the BaseApp.

When the player logs out, the BaseApp sends a logoff message to the database. This message contains the player's entity's properties, which may have been modified by the base (or cell) entity.

4.2.7. Reviver

Reviver is a watchdog process used to restart other processes that have failed, either because the machine they were running on has failed, or because the processes themselves have crashed or are unresponsive.

4.2.8. BWMachined

BWMachined is a daemon process that runs on each machine in the server cluster. It can be used to remotely start, stop, and locate server components. It also monitors CPU, memory, and network usage, and provides this information to network users.

This is how server components locate each other during startup, via a broadcast message. Its operation is similar to a CORBA Name Server.

This daemon must be running at all times on each machine in the server's cluster, and must be started as root. For installation instructions, see the document Server Installation Guide.

4.3. Use Cases

This section describes how some of the functionality of the server works. It is intended to give an introductory view of the components in the server, and how they relate to each other.

4.3.1. Server Startup

The important things about startup are the dependencies between processes and how different processes find one another.

As mentioned in “BWMachined” on page 40, BWMachined runs on each machine in the server cluster, and is responsible for monitoring the processes running on its machine.

When a process starts up, it registers itself with the daemon running on the local machine. The registration includes an interface name.

Many processes need to know the location of other processes.

For example, a CellApp needs to know the location of a CellAppMgr, so that it can register with it. So when CellApp starts up, it asks for the location of a CellAppMgr by sending a broadcast message on BWMachined's port, to which all BWMachined processes listen. If a BWMachined has a CellAppMgr (associated with the correct user), it sends the CellAppMgr's contact details as a response.

Note

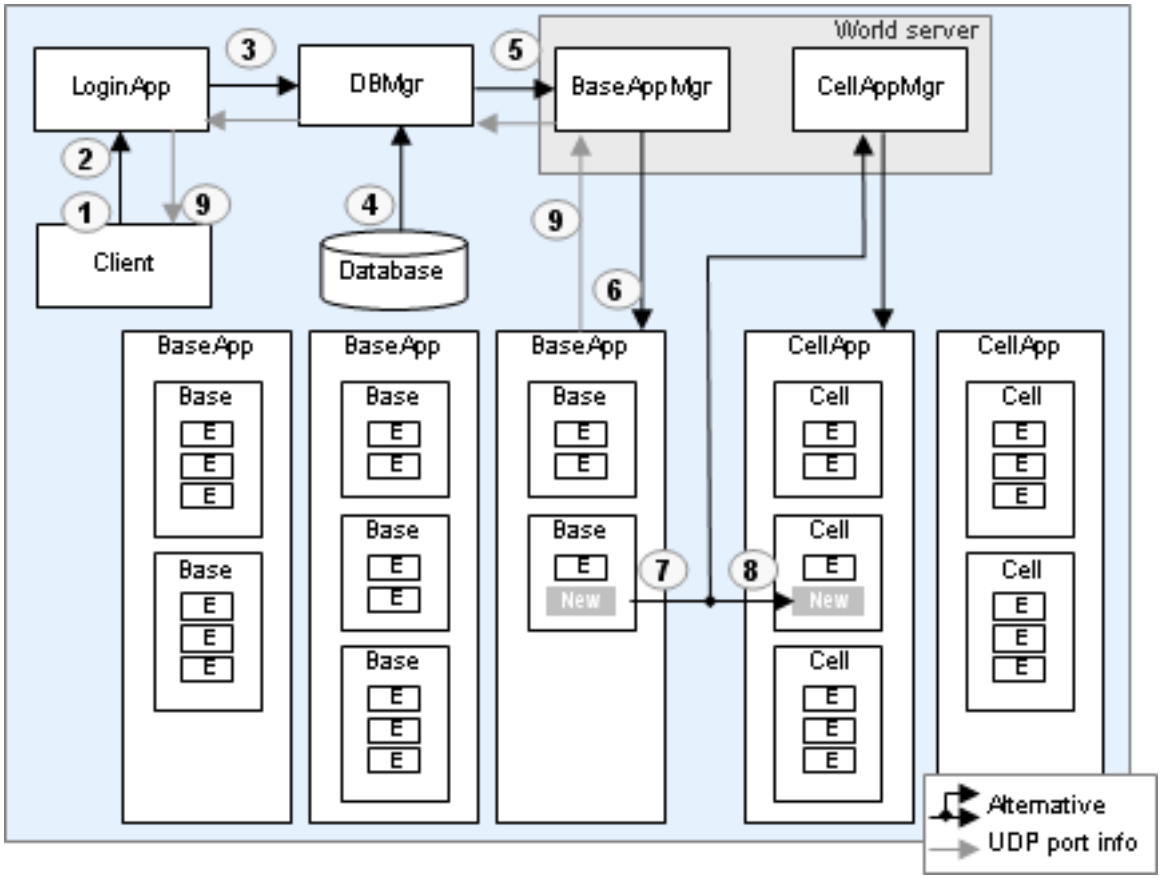
The BWMachined daemon should always be running on each machine in the server cluster.

4.3.2. Logging In

When a client starts up, it communicates with LoginApp (for more details, see “LoginApp” on page 16).

The login process follows the steps below:

1. Client sends a login request (for which it needs to know LoginApp's IP address).
2. While listening to its fixed (user-specified) port, LoginApp receives the request.
3. LoginApp forwards the request to DBMgr, so it can check if login details are valid.
4. DBMgr queries the database regarding login details.
5. If the details are valid, DBMgr instructs BaseAppMgr to create a new entity.
6. BaseAppMgr forwards entity creation request to the least loaded BaseApp.
7. BaseApp creates new proxy, which is an object derived from Base class.
8. The newly created proxy now may create the cell entity on a CellApp (this step might be delayed by the proxy until client selects a character, for example.).When creating the cell entity, the proxy may send a message via the CellAppMgr or directly to a CellApp
9. The UDP port of the proxy is returned to the client (via BaseAppMgr, DBMgr, then LoginApp).



Login process

4.3.3. Data From Clients

All communication from the client to the server is sent to the address of the client's proxy that was sent to it during login.

This communication uses Mercury (the BigWorld communication mechanism) on top of UDP. Once the proxy receives the packet, it forwards the messages (generally most, if not all) to the appropriate CellApp. This will then update the data of the appropriate entity. If the entity has any ghosts, these are also updated.

4.3.4. Data To Clients

Periodically, at 10 Hertz (configurable), the client's cell entity constructs a packet full of messages about the entities in the client's AoI. This packet is then sent via the proxy to the client.

4.3.5. Ghosting

Each client's entity keeps a priority queue of other entities in its AoI. One issue that arises is that not all these entities might be on the same cell. The solution to this is ghosting.

A ghost is a copy of an entity from an adjacent cell. The ghost copy contains all data that may be wanted when other entities are going through their priority queues to construct their update packets. If there is a possibility that an entity is within the AoI of another entity in a different cell, it must have a ghost in that cell. To achieve this, if an entity is within the AoI distance (or ghost distance) of a cell boundary, BigWorld creates a ghost of the entity on that cell.

When a real entity (*i.e.*, the master, non-ghost entity) is updated, it updates its ghosts as well, if any.

4.3.6. Changing Cells

As an entity moves around, it may actually leave the boundary of the cell it is currently on. When this occurs, the entity is moved to the relevant new cell. It then informs the base of its new address, so that the base can find it in the future (and in the case of a proxy, so that the proxy can continue to forward messages correctly).

4.3.7. Load Balancing

A CellApp machine, like any other machine, has a limit to the load it can handle. To avoid some CellApps getting overloaded, there is a mechanism to balance the load. In general, if a cell is overloaded, then it reduces its size, thus offloading some entities. If it is underloaded, it increases its size to assume control of more entities.

Chapter 5. Server Components

This section describes the various components on the server side of BigWorld Technology and how they work together to support the game environment.

5.1. CellApp

5.1.1. Cell Application and Cells

A Cell Application, or CellApp, is the actual process or executable running on a CellApp machine.

A CellApp manages various cells, or instances of the Cell class. A cell is a geometric part of a large space, or the entirety of a small space.

BigWorld divides large spaces into multiple cells in order to share the load evenly. It typically allocates one large cell to each CellApp. When handling smaller spaces, such as dynamically created mission ones, BigWorld typically allocates several cells to each CellApp.

In BigWorld, the CellApp machines are intended to run only BWMachined and one CellApp per CPU, and nothing else.

5.1.2. Entities

CellApps are concerned with a generic type, the entity. Each CellApp maintains a list of the real and ghost entities within its reach. The CellApp also maintains a list of any real entities that need to be updated periodically (*e.g.*, 10 times per second), and those that maintain an AoI.

Each entity understands the messages that are associated with its type. The CellApp delivers entity messages to the appropriate entity, and it is up to it to interpret them. The way this is implemented is by having every entity include a pointer to an entity type object. This object has information related to the entity's type, such as the messages that the type can handle and the script associated with it.

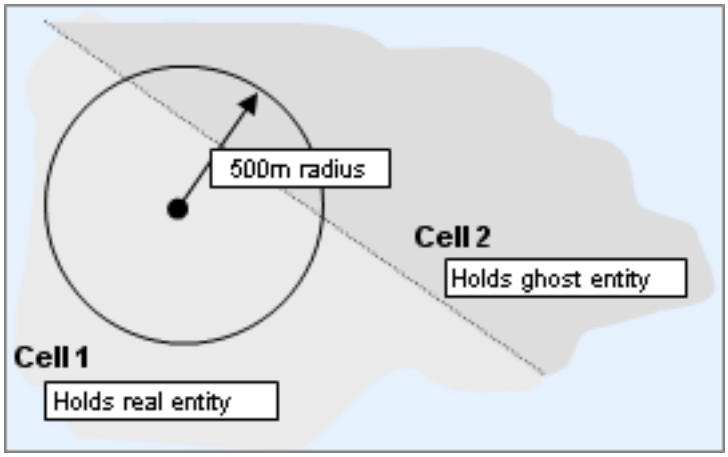
Every entity type has it own script file. Scripts execute in the context of a particular entity (*i.e.*, they have a this or self), and have access to the data that other entities choose to make available (server and client data), as well as the basic members of every entity (`id`, `position`, ...).

Scripts are responsible for handling messages sent to an entity.

When any server or client data is changed, the changes are automatically forwarded to interested entities.

5.1.3. Real and Ghost Entities

In order to efficiently update its client, each avatar keeps a list of entities in its AoI, which is typically a circle with a radius of 500m. One issue that arises is that not all entities might be controlled by the same cell.

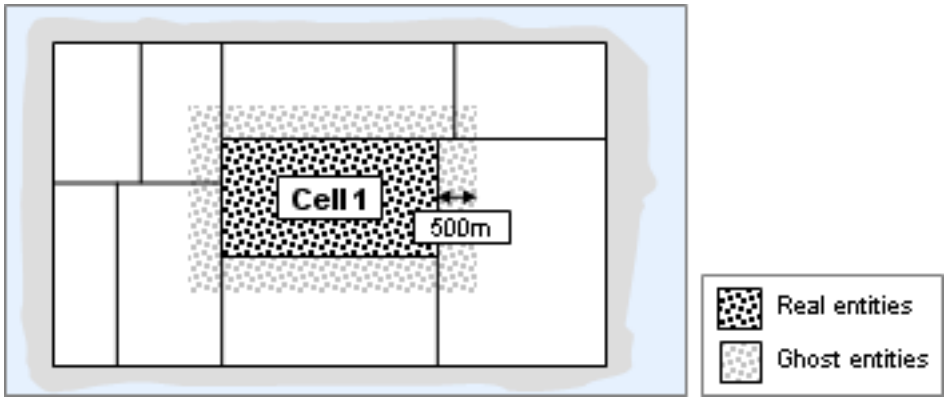


Entity's AoI of crossing cell boundaries

The solution to this is ghosting. A ghost is a copy of an entity from a nearby cell. The ghost copy contains all entity data that may be needed when avatars consume their priority queues to construct their update packets.

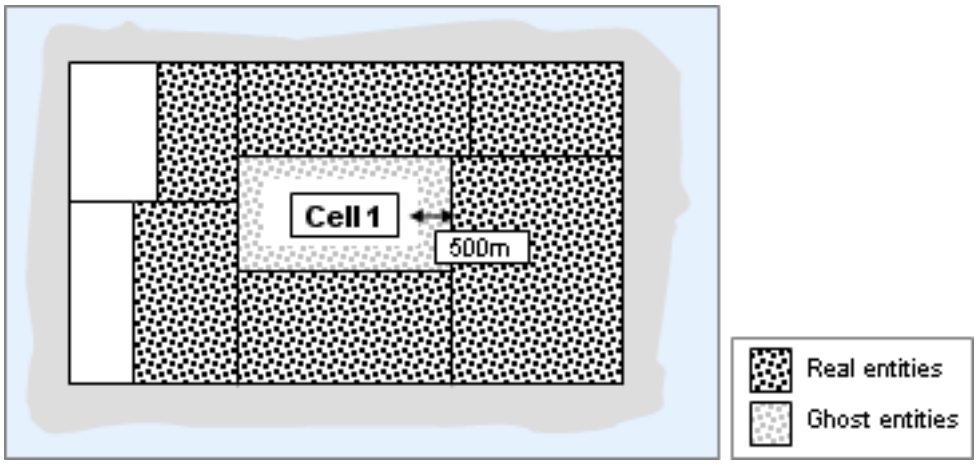
However, it would be very inefficient to have each entity individually determine which entities on adjacent cells should be ghosted. The solution is to have cells generically manage ghost entities. If an entity is within the ghost distance of a cell boundary, the cell creates a ghost of the entity on that nearby cell.

The term real entity is introduced to distinguish between the master representations and the (imported) ghosts. The figure below shows all entities maintained by cell 1.



Cell 1 maintains ghosts of entities controlled by adjacent cells

Once a second, a cell goes through its real entities to check if it should add or delete ghosts for them on neighbouring cells. It sends messages to the neighbour to add and remove the ghosts.



Cell 1 ensures that adjacent cells maintain ghosts of its real entities

5.1.4. Transitioning Between Spaces

The simplest method to transition between different spaces is simply to 'pop', *i.e.*, to remove the entity from the old space and recreate it in the new one. This is simple on both client and server.

A somewhat more sophisticated technique is to have an enclosed transition area such as a lift, which is precisely duplicated in both spaces. After the player enters the transition area, and it becomes enclosed, the developer can 'pop' the player out of the old space and into the duplicate copy of the area in the new space. The client needs to transform the player's position into the new coordinate system, discard knowledge of entities in the old space, and start building up knowledge of entities in the new space. Once the client has been updated and the position filters of the new entities are sufficiently filled, the player can leave the transition area (lift door opens) and continue. Much of this is automated by the CellApp.

5.1.5. Witness Priority List

Whenever an entity has a client attached to it, a sub-object known as a witness is associated to the entity, in order to maintain the list of entities in its AoI. The witness builds update packets to send to its client, and it must send the most important information as a priority. The client requires the position and other information of other entities that are closest to it to be the most accurate and current. Closer entities should be updated more frequently. To achieve this, the witness keeps the list of entities in its AoI as a priority queue.

To construct a packet to be sent to the client, relevant information about the entities on the top of the list is added to the packet until the desired size is reached. The information can include position and direction, as well as events that the entity has processed since the last update. For more details, see "Event History" on page 23 .

Closer entities receive more frequent updates, and get a greater share of the available bandwidth.

BigWorld throttles downstream bandwidth according to both the size of update packets, and their frequency. These parameters are specified in the configuration file `<res>/server/bw.xml`.

5.1.5.1. Event History

Each entity keeps a history of its recent events. The history includes both events that change its state (such as weapon change), as well as actions that do not (such as jumping and shooting). Frequent actions, such as movement, which affect volatile data, are not kept in the history list.

When an entity consumes its priority queue to construct an update packet, it searches through new events of the top entities (*i.e.*, the ones closes to it), and adds all messages it is interested in. This requires keeping a timestamp (actually a sequence number) for the last update of each entity in the priority queue.

The history of these types of events is fairly short (around 60 seconds) since we expect that each entity in the priority queue will be considered (roughly) once every 30 seconds in the worst case scenario.

5.1.5.2. Level of Detail

The priority queue handles the data throttling for continuous information. For events, we still need to handle the case where there are too many events filling up available downstream bandwidth to the client.

When there are many entities in an entity's AoI, information about each one is sent less often, but the total number of events remains the same.

To address this, BigWorld uses a Level of Detail (LOD) system for handling event-based changes. Its principle is that the closer an entity is to the avatar, the more detail it should send to its client. For example, when a player initially comes within the avatar's AoI, the avatar does not need to know all details about the other player. The visible inventory may only be required when within, say, 100 metres. Finer details, such as a badge on clothing, might only be needed within 20 metres.

- **Non-state-changing events**

  In the description of each entity type, there is a description of all messages (non-state-changing events) it can generate, along with the priority associated with each one.

  When an avatar adds information about an entity, it only adds messages with a priority greater than the avatar's current interest in the entity. Therefore, messages will be ignored depending on the distance between the avatar and the entity.

  For example, there might be a type of chat heard only within 50 metres, or a jump seen only within 100 metres.

- **State-changing events**

BigWorld cannot ignore state-changing events that occur out of range, because if the entity subsequently does get close enough, the client will have an incorrect copy of the entity's state.

For example, the client may be interested in the type of weapon that an entity is carrying, but only when it is within 100 metres. If the entity changes its weapon outside this range, then this information is not sent to the client as yet. BigWorld will only send it if the entity comes within range.

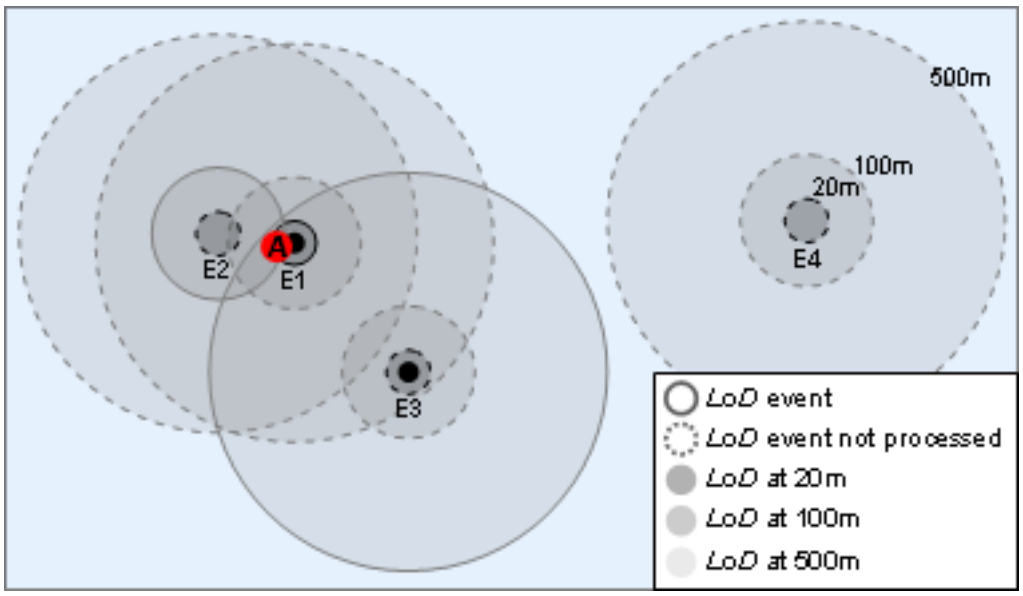
Unlike non-state-changing events, in which the priority level can have any value, state-changing events have a discrete number of state LODs specified by the developer. These can be thought of as rings around the client. Each property of an entity's type can be associated with one of these rings. For an example, see the properties `<SeatType>` and `<LODLevels>` in the file `<res>/scripts/entity_defs/Seat.def` on “Example Entity Definition File” on page 25 .

For example, supposed an avatar A is surrounded by four entities of the same type, with LOD levels at 20m, 100m, and 500m. It will process each entity according to its distance, as described in the table below:

Entity	Distance	Process LOD events		
		20m	100m	500m
E1	15m	✓	✓	✓
E2	90m	X	✓	✓
E3	400m	X	X	✓
E4	1,000m	X	X	X

LOD events processed by avatar A

The diagram below illustrates the example:



LOD levels of entities surrounding avatar A

When an entity with an associated witness comes within an *LOD* ring of another entity, any state associated with the ring that has changed *since they were last this close* is sent to the client. BigWorld uses timestamps to achieve this. A timestamp is stored with each property of each entity. This timestamp indicates when that property was last changed. Also, for each entity in a witness' *AoI*, a timestamp corresponding to when that entity last left that level is kept for each *LOD* ring.

By itself, this does not allow us to throttle this data. To achieve this, we only need to apply a multiplying factor to the interested level of the avatar when it is calculated for other entities. Multiplying by a factor less than one has the effect of reducing the amount of data sent.

5.1.6. Scripting and Entities

The following sub-sections describe how to use scripting for entities.

5.1.6.1. Entity Classes

An entity class describes a type of entity. The list below describes its component parts:

- **An XML definition file**

  `<res>/scripts/entity_defs/<entity>.def`

- **A Python cell script**

  `<res>/scripts/cell/<entity>.py`

- **A Python base script**

  `<res>/scripts/base/<entity>.py`

- **A Python client script**

  `<res>/scripts/client/<entity>.py`

The XML definition file can be considered the interface between the Cell, Base and Client scripts. It defines all available public methods, and all properties of an entity. In addition, it defines global characteristics of the class, such as:

- Whether the entity requires volatile position updates.
- How the entity is instantiated on the client.
- The base class of the entity, if any.
- The interfaces implemented by the entity, if any.

5.1.6.2. Example Entity Definition File

The following is an example entity definition file `<res>/scripts/entity_defs/Seat.def`.

```
<root>
  <Properties>
    <seatType>
      <Type>         INT8          </Type>
      <Flags>        OTHER_CLIENT  </Flags>
      <Default>      -1            </Default>  <!-- See Seat.py -->
      <DetailLevel>  NEAR          </DetailLevel>
    </seatType>
    <ownerID>
      <Type>         INT32         </Type>
      <Flags>        OTHER_CLIENT  </Flags>
      <Default>      0             </Default>
    </ownerID>
    <channel>
      <Type>         INT32         </Type>
      <Flags>        PRIVATE       </Flags>
    </channel>
```

```
  </Properties>

  <ClientMethods>

    <clientMethod1>
      <Arg>            STRING  </Arg>  <!-- msg -->
      <DetailDistance> 30      </DetailDistance>
    </clientMethod1>

  </ClientMethods>

  <CellMethods>

    <sitDownRequest>
      <Exposed/>
      <Arg>      OBJECT_ID  </Arg>  <!-- WHO -->
    </sitDownRequest>

    <getUpRequest>
      <Exposed/>
      <Arg>      OBJECT_ID  </Arg>  <!-- WHO -->
    </getUpRequest>

    <ownerNotSeated>
    </ownerNotSeated>

    <tableChat>
      <Arg>      STRING    </Arg>  <!-- msg -->
    </tableChat>

  </CellMethods>
  <LODLevels>
    <level>  20   <hyst>   4  </hyst> <label> NEAR   </label>  </level>
    <level>  100  <hyst>  10  </hyst> <label> MEDIUM </label>  </level>
    <level>  250  <hyst>  20  </hyst> <label> FAR    </label>  </level>
  </LODLevels>

</root>
```

Entity definition file <*res*>/scripts/entity_defs/Seat.def

5.1.6.3. Properties

As shown in the listing in the previous sub-section, all properties are defined in the XML file, along with their data type, an optional default value, flags to indicate how they should be replicated, and an optional tag `DetailLevel` for LOD processing.

All cell entity properties need to be defined, even if they are private to the class. This is because CellApps need to offload entities to other CellApps, and having the data types of all properties defined allows the data to be transported as efficiently as possible.

When a Python script modifies the value of a property, the server does the work of propagating this property change to the correct places.

The following flags are used in the XML definition to determine how each property should be replicated[1]:

- **`ALL_CLIENT`**

  Implies the `CELL_PUBLIC` flag

[1] For more details, see the document Server Programming Guide's chapter *Properties* → "Data Distribution".

Only relevant for entities with an associated client, i.e., a player entity

Properties that are visible to all nearby clients, including the owner. Corresponds to setting both the `OWN_CLIENT` and `OTHER_CLIENT` flags.

- **`ALL_CLIENTS`**

  Same as `ALL_CLIENT`. This enumeration is deprecated.

- **`BASE`**

  Properties that are used on bases.

- **`BASE_AND_CLIENT`**

  Properties that are used both on bases and clients. Corresponds to setting both `OWN_CLIENT` and `BASE` flags.

- **`CELL`**

  Same as CELL_PUBLIC. This enumeration is deprecated.

- **`CELL_AND_OWN`**

  Implies the `CELL_PUBLIC` flag

  Same as CELL_PUBLIC_AND_OWN. This enumeration is deprecated.

- **`CELL_PRIVATE`**

  Properties that contain state information that is internal to an entity. They are available on to the owner, and only on the cell. They will not be ghosted, and as such, entities on other cells will not be able to see them.

- **`CELL_PUBLIC`**

  Properties that are visible to other entities on the server. They will be ghosted, and any other nearby entity will be able to read them, whether they are in the same cell or not (as long as they are within the AoI distance).

- **`CELL_PUBLIC_AND_OWN`**

  Properties that are available to other entities on the cell, and to this one on both the cell and the client.

- **`CLIENT_ONLY`**

  Properties that are used only on clients.

- **`GHOSTED`**

  Same as CELL_PUBLIC. This enumeration is deprecated.

- **`GHOSTED_AND_OWN`**

  Same as CELL_PUBLIC_AND_OWN. This enumeration is deprecated.

- **`OTHER_CLIENT`**

  Implies the `CELL_PUBLIC` flag

  Same as OTHER_CLIENTS. This enumeration is deprecated.

- **`OTHER_CLIENTS`**

  Implies the `CELL_PUBLIC` flag

  Properties that are visible to nearby clients, but not to the owner.

- **`OWN_CLIENT`**

  Only relevant for entities with an associated client, i.e., a player entity

  Properties that are only visible to the client who owns the entity.

- **`PRIVATE`**

  Same as CELL_PRIVATE. This enumeration is deprecated.

When a property changes, the server uses the flags OWN_CLIENT and ALL_CLIENTS to determine if an update should be sent to the entity's own client, and the flags OTHER_CLIENTS and ALL_CLIENTS to determine if an update should be sent to the other clients.

If a server script modifies a property marked as ALL_CLIENTS, then that change will be replicated to:

- The owner of the entity Because it corresponds to also setting the OWN_CLIENT flag.
- All nearby clients Because it corresponds to also setting the OTHER_CLIENT flag.
- All ghosts on adjacent cells Because it implies the GHOSTED flag.

Note

When a client modifies a shared property, the change is not propagated to the server. All communication from the client to the server must be done using method calls.

Property definitions can also include the tag `DetailLevel`. This is used for data LOD processing. For an example, see the property `SeatType` defined in the example definition file on "Example Entity Definition File" on page 25. For details on LOD processing, see the document Server Programming Guide's section *Proxies and Players* → "Entity Control".

5.1.6.4. Built-in Properties

In addition to the properties defined in the XML file, cell entity scripts also have access to several built-in properties provided by the cell scripting environment. These include:

- **`id` (Read-only)**

  Integer ID of the entity.

- **`spaceID` (Read-only)**

  ID of the space that the entity is in.

- **`vehicle` (Read-only)**

  Some other entity (or None) that the entity is riding upon.

- **`position` (Read/write)**

  Position of the entity, as a tuple of 3 floats.

- **`direction` (Read/write)**

Direction of the entity, as a tuple of roll, pitch, and yaw.

- **`isOnGround` (Read/write)**

  Integer flag, containing 1 if the entity is on the ground, or 0 otherwise.

It is possible to move an entity by changing its position property. However, for continuous movement, the method `moveToPoint` is recommended. The `spaceID` and `vehicle` properties are influenced by the `teleport` and `boardVehicle` methods, respectively.

5.1.6.5. Methods

Like properties, methods are defined in the entity type's XML definition file (named `<res>/scripts/entity_defs/<entity>.def`). There are different sections for client, cell, and base methods, and each method contains its name and its list of arguments.

Cell and base methods can also have an optional `<Exposed/>` tag. This indicates whether the client can call this method. Exposed methods on the cell have an implicit argument, `source`, which is the ID of the entity associated with the client that called the method.

Method definitions can also have a `<DetailDistance>` attribute which is used in relation to Level of Detail filtering.

5.1.6.6. Calling Client methods

A client has references to all entities that are within its AoI. A cell component of an entity can call methods on other client entities (including its own) that exist within this AoI.

Four properties are available to the cell script, to facilitate calling client methods. These are:

- **`ownClient` (or `client`)**
- **`otherClients`**
- **`allClients`**
- **`clientEntity`**

A method called on one of these objects will be delivered to the indicated clients.

For example, to call the chat method on the client script for this object, for all nearby clients:

```
self.allClients.chat( "hi!" )
```

5.1.6.7. Built-in Methods

The cell script has access to numerous built-in script methods, some of which are described in the list below:

- **`destroy`** — Destroys the entity.
- **`addTimer`** — Adds an asynchronous timer callback.
- **`moveToPoint`** — Moves the entity in a straight line towards a point.
- **`moveToEntity`** — Moves the entity towards another entity.
- **`navigate`** — Moves the entity towards a point, avoiding obstacles.
- **`cancel`** — Cancels a controller (timers, movement, etc...).

- **`entitiesInRange`** — Finds entities within a given distance of the entity.

5.1.6.8. Controllers

A controller is a C++ object associated with a cell entity. It normally performs a task that would be inefficient or difficult to do from script. It also generally has state information, which must be sent across the network if the entity is offloaded to another cell.

Examples of currently implemented controllers are:

- **`TimerController`** — Provides asynchronous timed callbacks.
- **`MoveToPointController`** — Moves the entity to a position, at a given velocity.

Controllers are normally created by a script call, such as *`entity`*`.moveToPoint`, or *`entity`*`.addTimer`. Any script call that creates a controller should return a controller ID. This is an integer that uniquely identifies the controller, and can later be used to destroy it. Calling *`entity`* `.cancel`, and passing in the controller ID can destroy any controller. Note that the cell automatically destroys all controllers when an entity is destroyed.

Since controllers normally perform operations asynchronously, they notify the script object of completion by calling a named script callback. For example, the `TimerController` always calls the `onTimer` event handler, and the `MoveToPointController` always calls the `onMove` event handler.

5.1.6.9. Inheritance

An entity class may be derived from another entity class. It receives all properties and methods of its base class, and adds its own interfaces. The `<Implements>` section in the `.def` files can be used to describe multiple interfaces.

In addition, it is possible for a derived class to appear as a base class when instantiated on the client. This is useful in circumstances where the client does not need to be aware of all extra methods and properties implemented for the derived class on the server; the client can just treat derived class entities as base class entities. This also allows new classes to be derived from the base class on the server, without the client being updated.

5.1.6.10. Example Script File

The following is an example cell entity script file for a `Seat` object:

```
"This module implements the Seat entity."
import BigWorld
import Avatar
# --------------------------------------------------------------------
# Section: class Seat
# --------------------------------------------------------------------
class Seat( BigWorld.Entity ):
  "A Seat entity."
  def __init__( self ):
    BigWorld.Entity.__init__( self )

  def sitDownRequest( self, source, entityID ):
    if self.ownerID == 0 and source == entityID:
      self.ownerID = entityID
      BigWorld.entities[ self.ownerID ].enterMode(
        Avatar.Avatar.MODE_SEATED, self.id, 0 )
      if self.channel:
        try:
          channel = BigWorld.entities[ self.channel ]
          channel.register( self.ownerID )
        except:
```

```
        pass

  def getUpRequest( self, source, entityID ):
    if self.ownerID == entityID and source == entityID:
      if self.channel:
        try:
          channel = BigWorld.entities[ self.channel ]
          channel.deregister( entityID )
        except:
          pass
      BigWorld.entities[ self.ownerID ].cancelMode()
      # The Avatar's cancelMode() method will in-turn call this
      # Seat's ownerNotSeated() method to release itself.

  def ownerNotSeated( self ):
    self.ownerID = 0

  def tableChat( self, msg ):
    if self.channel:
      channel = BigWorld.entities[ self.channel ]
      assert( channel )
      assert( self.ownerID )
      channel.tellOthers( self.ownerID, "[local] " + msg )
```

Cell entity script — *<res>*`/scripts/cell/Seat.py`

5.1.7. Directed Messages

Not all events are propagated via the event history. If an event is destined for a specific player, or a small number of players, then it can be delivered almost immediately in its next packet.

An example is when a player talks to a targeted avatar. When the message arrives, it can be delivered to the desired avatar immediately, which in turn can add it to its next bundle to be delivered to the client.

5.1.8. Forwarding From Ghosts

Each ghost knows its real cell, so that it can forward messages to its real version to be processed.

Most messages are of the pull variety, as in when another player jumps — it is up to the avatar (via the priority queue) to decide if and when to send this to its client.

A minority of the messages is of the push variety, as in when player A wants to shake hands with player B. To do this, client A sends a handshake request to its cell, which in turns makes a request to avatar B on the same cell. Avatar B may be a ghost, in which case it forwards the request to its real representation. This communication happens automatically and transparently.

The mechanism of forwarding messages from ghosts to their reals is also used to circumvent the synchronisation problem that occurs when entities change cells. Normally the base entity sends messages directly to the real, but it may temporarily send messages via a ghost when the cell entity is in transit.

5.1.9. Offloading Entities

Approximately once a second each cell checks whether to offload any entities to other cells. Each entity is considered against the boundary of the current cell. The boundary is artificially increased (by about 10 metres) to avoid hysteresis. If an entity is found to be outside the boundary of the current cell, it is offloaded to the most appropriate one.

When an entity is offloaded, the real entity object is transformed into a ghost entity, and vice-versa when an entity is loaded.

5.1.10. Adding and Removing Cells

When a cell is added to a large multi-cellular space during runtime, it is done by gradually increasing its area, in order to avoid a large spike in entities trying to change cells. Similarly for removing a cell, its area slowly decreases until all entities are gone.

For more details, see “Adding and Removing Cells” on page 34 .

5.1.11. Load Balancing

Cell boundaries are moved in order to adjust the proportion of the server load that an individual cell is responsible for.

The basic (simplified) idea is that if a cell is overloaded in comparison to other cells, then its area is reduced. Conversely, if it is underloaded, then its area is increased.

5.1.12. Physics

Because of factors such as latency inherent in traversing the Internet, large number of desired players, and bandwidth limits, it is difficult (if not impossible) to achieve a convincing game where full collision handling is attempted.

For example, consider running through a crowd. Because of the inaccuracy in player positions resulting from latency and bandwidth limits, their server positions might have them colliding, when in fact they are avoiding each other on the client. Even with full collisions, it is still not clear that this is good from a game play point of view, since it might become too difficult to walk through the crowd.

For these reasons, BigWorld implements an entity-to-entity collision detection system only on the client side, whereby clients are bumped aside when they try to occupy the same space. There is potential to extend this to send messages to the server when the collision is severe, *e.g.*, the server can check the situation and maybe change each avatar's position/velocity.

Collisions against the static scene however, may be checked by the server. The algorithm works by ensuring that player entities travel through well-defined portals, and not through walls, floors, and ceilings. This is enabled by setting the topSpeed property to greater than zero (the entity's speed is also checked).

For more details, see the document Server Programming Guide's section *Proxies and Players* → “Entity Control”.

5.1.13. Navigation System

The key features of the navigation system are:

- Navigation of both indoor and outdoor chunks.
- Seamless transition between indoor and outdoor regions.
- Dynamic loading of navpoly graphs.
- Paths caching, for efficiency.

For details, see the document Server Programming Guide, section *Entities and the Universe* → “Navigation System”.

5.1.14. Range Triggers and Range Queries

There is often the need to trigger an event when an entity (of a specific type) gets close enough to another. These are called Range Triggers.

There is also the need to find all entities that are in a given area. This is called a Range Query.

To support these functions, each cell maintains two lists of all its entities, sorted by X and Z.

To perform a range query, the software searches just one of these lists to the distance of the query range, and then selects the entities geometrically in range.

To perform a range trigger, the entity that is interested in a particular range inserts high and low triggers in both the X and Z lists. When the entity moves, the lists and the triggers are updated. When other entities move, the lists are updated. If an entity crosses a trigger, or a trigger crosses an entity, then the software checks the other dimension to test whether it is actually a trigger event.

For the majority of situations this algorithm has been found to be very effective.

5.1.15. Fault Tolerance

The CellApp is designed to be fault tolerant. A CellApp process can stop suddenly, or its machine can become disconnected from the network, with little noticeable impact on the server.

Each entity on a CellApp is periodically backed up to its base entity. Should a CellApp become unavailable, the CellAppMgr ensures that each space still has at least one cell. The base entities detect if their cell entities have been lost, and restore them to an alternate cell of their space. Script methods are available to ensure that restored entities are consistent with the game world.

For more details, see the document Server Operations Guide's chapter *Fault Tolerance*. For details on the implementation of CellApp fault tolerance on code level, see the document Server Programming Guide's chapter *Fault Tolerance*.

5.2. CellAppMgr

The CellAppMgr is responsible for:

- Maintaining the correct adjacencies between cells.
- Adding new avatars to the correct cell.
- Acting as a global entity ID broker.

5.2.1. CellApp Registration

When a new CellApp is started, it finds the location of the CellAppMgr by broadcasting a message to all BWMachined daemons.

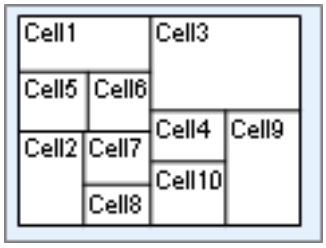
If a BWMachined process has a CellAppMgr on its machine, then its address is returned to the CellApp. Once the CellApp has the location of the CellAppMgr, the CellApp registers itself with it. Next time the CellAppMgr needs to create more cells, the new CellApp will be considered as a potential host for it.

Similarly, when the CellApp stops, it deregisters itself with the CellAppMgr, after having all its cells gradually removed.

5.2.2. Load Balancing

BigWorld dynamically balances the load of cells for all spaces, from small single-cell spaces to potentially huge multi-cell ones.

BigWorld divides large spaces into cells using a geometric tessellation. The example tessellation provided below illustrates the current algorithm.



Example of typical space

The amount of Euclidean space covered by a cell is determined by whatever load can be supported by its CellApp. This may vary with either CPU or entity density.

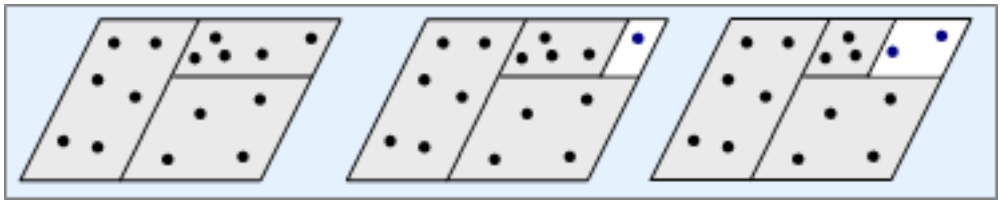
The details of the current tessellation scheme are beyond the scope of this document.

5.2.3. Adding and Removing Cells

In order to support smooth start up, shutdown, and scalability, BigWorld needs to be able to add and remove cells in an orderly fashion. To smoothly add a new cell to the system, BigWorld slowly increases the area managed by the new cell.

At first, no entity is maintained by the new cell, since none lies within its initial area. As the cell's area increases, entities migrate to it at a controlled rate. The reverse is done to remove cells.

The image below shows a potential mechanism for achieving this, in line with the tessellation example in topic “Load Balancing” on page 33 .



Inserting a new cell into the system

It is the CellAppMgr's job to divide the space, based on reports by the CellApps of their current load and areas of space they are managing. CellAppMgr uses this information to decide whether a CellApp should adjust its area of responsibility, or a new cell be created or an existing one be removed. Whatever is the outcome, it will communicate with the CellApp.

Apart from knowing its own area of responsibility, every CellApp also has some basic knowledge (address and area of responsibility) of the other CellApps sharing the same space (this information is provided to them by CellAppMgr). A CellApp can decide to establish communication channels with any number of existing CellApps (usually with its neighbours) to either offload entities, or to mirror entity data on them (based on entity's location).

5.2.4. Adding an Entity

The CellAppMgr can easily determine which cell to add a new entity to, since it controls the tessellation of the space.

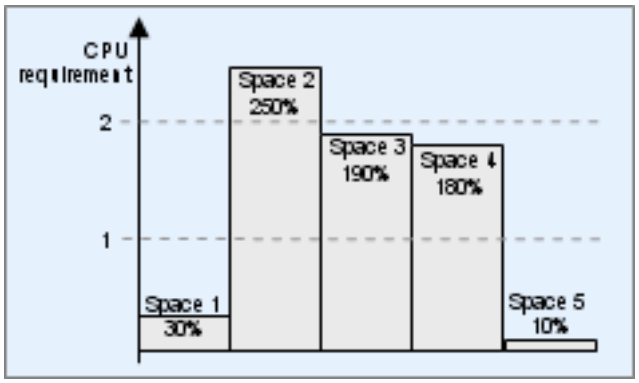
However, a new cell entity will often bypass the CellAppMgr (thereby reducing its load) if it knows of another entity in the same space, close to where it wants to be created. In that case it will simply create itself on the same CellApp of the existing entity.

5.2.5. Load Balancing for Multiple Spaces

In case of multiple spaces, BigWorld dynamically and continuously balances their load, from large to very small, using a single algorithm of optimal fitting.

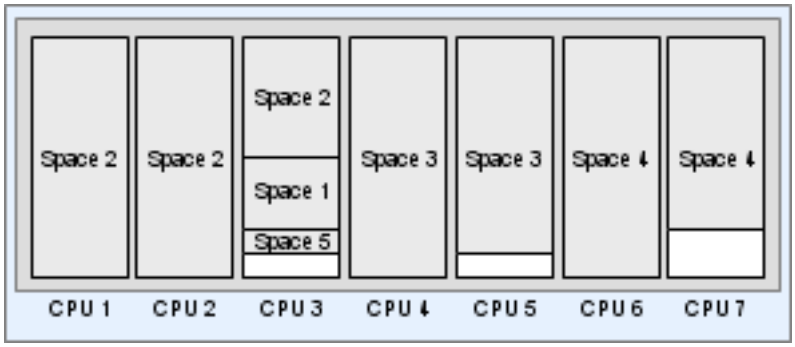
The basic goal of the algorithm is to allocate sufficient processing power to each space, and minimise the distribution of processing of each one to as few CellApps (machines) as possible. Where a space must be distributed over more than one machine, an algorithm is used to break up that space into multiple cells.

The algorithm is based on the current CPU requirement for each space. In the example below, there are five spaces, three of which require more than one CPU. For this example, all available CPUs are assumed to be of equal power.



CPU requirement per space

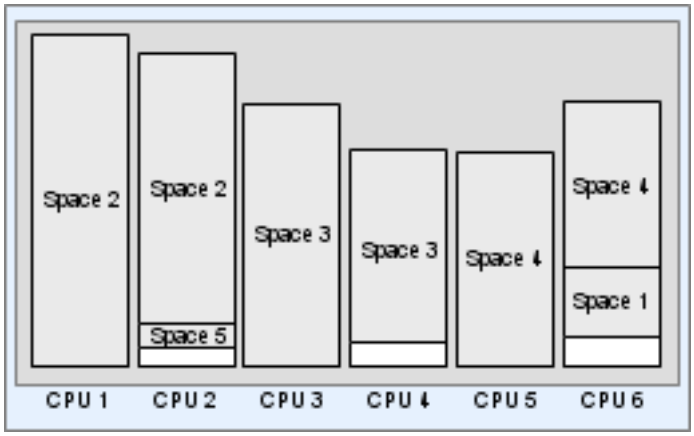
Working from the largest CPU consumer to the smallest, BigWorld might distribute the load across seven identical CPUs, as follows:



CPU allocation per space (identical CPUs)

BigWorld can profile CPUs individually to make optimal use of them

In the diagram below, the same spaces are allocated to CPUs with different power (more powerful CPUs are represented by taller boxes.):



CPU allocation per space (different CPUs)

5.2.6. Fault Tolerance

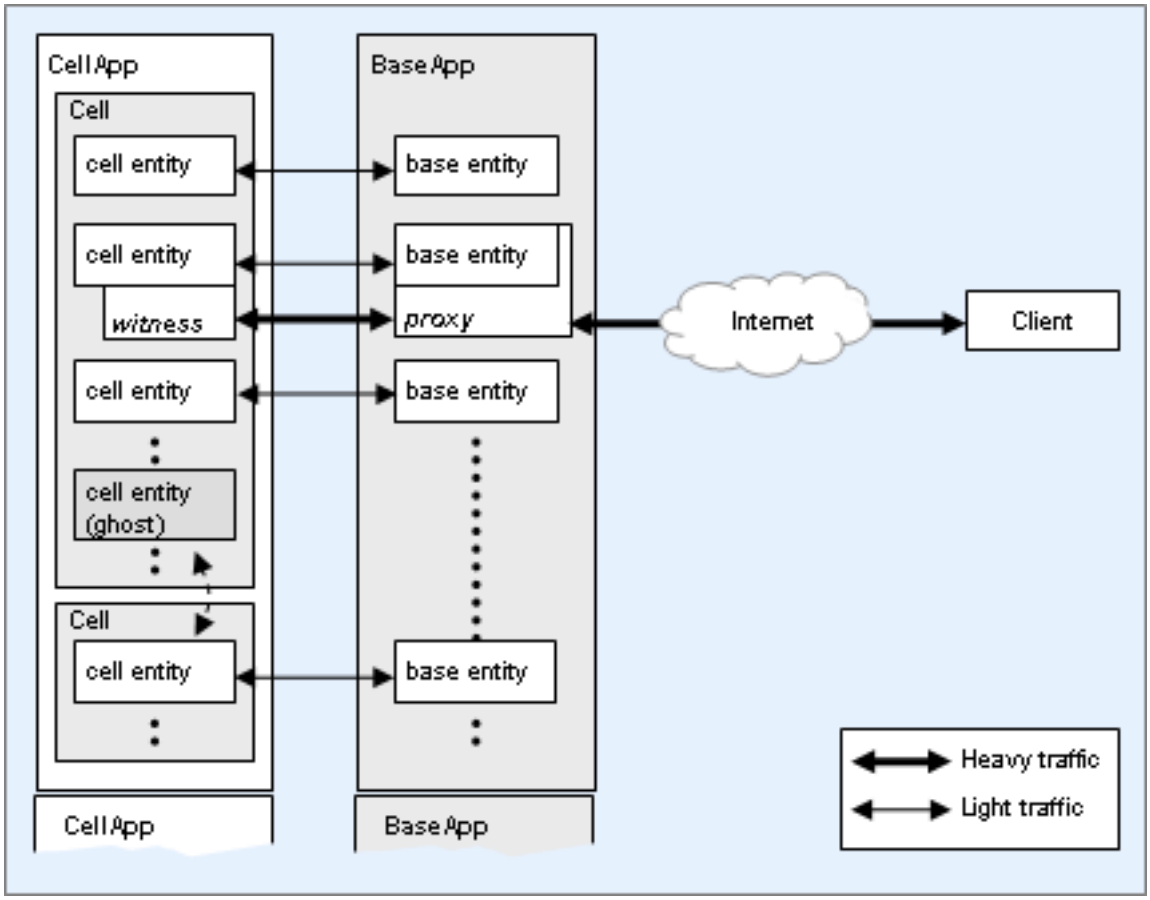
In case of failure, a Reviver restarts the CellAppMgr. It finds all CellApps, and queries them for their cells and the area they cover.

From this point on, it can keep maintaining the cells, CellApps, and spaces, and is able to add new entities to the correct cell. It can continue in its role as a space ID and entity ID broker by recovering data from stored checkpoints, or ID ranges maintained by the cells.

For more details, see the document Server Operations Guide's chapter *Fault Tolerance*. For details on the implementation of CellApp fault tolerance on code level, see the document Server Programming Guide's chapter *Fault Tolerance*.

5.3. BaseApp

The BaseApp manages entity bases and proxies. A proxy is a specialisation of an entity base.



BaseApp managing bases and proxies

5.3.1. Proxies

When a player logs into the server, a proxy is created on a BaseApp, which may decide to create an entity in a cell on a CellApp. The BaseAppMgr creates the proxy on the least loaded BaseApp.

Besides the LoginApp, the machines running the BaseApps are the only ones that need to be connected directly to the Internet. The proxy gives the client a fixed object on a fixed IP and port to talk to. The LoginApp returns this IP address to the client after the proxy has been created.

When the client wants to send messages to its entity on the cell, it first sends it to the proxy, which then forwards it to the cell. This isolates the client from cell switches. To achieve this, the proxy needs to know

the address of the CellApp that its associated entity is on. Whenever the cell entity changes CellApps, it communicates with the proxy.

The cell entity also communicates with its client via the associated proxy. The main benefit of this approach is that it allows the proxy to be responsible for handling message reliability. Other benefits include shielding the CellApps from talking directly to the Internet, and limiting the addresses that the client receives data from.

The proxy can also aggregate messages from multiple sources and deliver them to the client as one packet. This may include messages from the proxy itself, or other objects, such as team bases. It is up to the proxy to allocate bandwidth to each source that communicates to the client.

5.3.2. Bases

Bases have the following characteristics:

1. A base can store entity properties that do not need to travel around on the cells — a good example is storing a player's inventory on the base, and active item(s) on the cell.
2. A cell entity (or any other server object) can talk to a base entity when it does not know which cell the entity is currently on. The base acts as an anchor point.
3. Some properties are more efficiently kept on the base, since they are accessed (or subscribed to) by 'remote' objects.
4. A good example is a team chat system.
5. A base can be an entity that has no sense of location, such as entity controlling world events.

The proxy is a special type of base. It has an associated client and controls messages to and from it.

Like entities on the cells, bases have a script associated with them. Thus, it is possible to implement different base types in script only. A typical example would be a group chat system.

5.3.3. Fault Tolerance

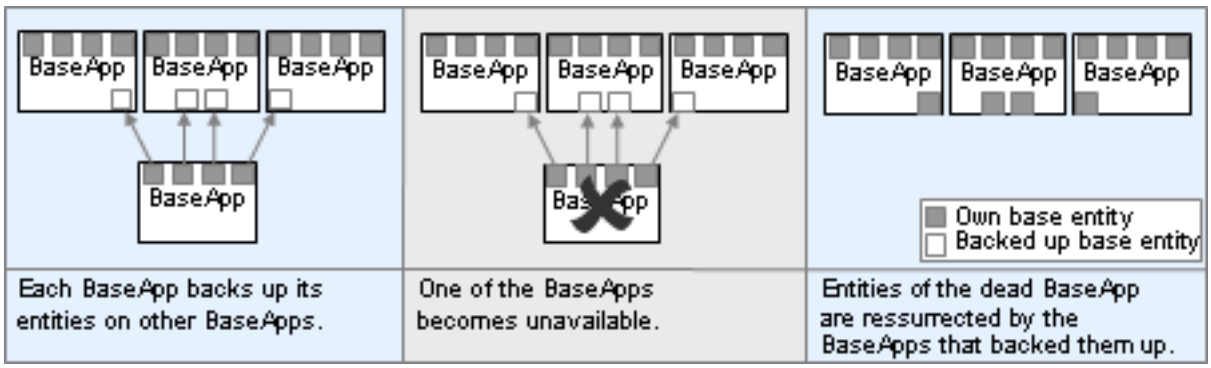
The BaseApp supports a scheme for fault tolerance where each BaseApp backs up its entities on other (more than one) regular BaseApps.

For more details, see the document Server Operations Guide, chapter *Fault Tolerance*. For details on the implementation of BaseApp fault tolerance on code level, see the document Server Programming Guide, chapter *Fault Tolerance*.

Each BaseApp backs up its entities across a number of other real BaseApps. Each BaseApp is assigned a set of other BaseApps and a hash function from an entity's ID to one of these backups.

Over a period of time, all entities of each BaseApp will be backed up on other regular BaseApps. This process constantly runs, making sure that new entities are backed up.

If a BaseApp dies, each of its entities is restored on the BaseApp that was backing it up.



Distributed BaseApp Backup — Dead BaseApp's entities can be restored to Backup BaseApp

There is a possibility of base entities that were previously on the same BaseApp ending up on different BaseApps (which places limits on scripting).

In addition, any attached clients will be disconnected on an unexpected BaseApp failure, requiring a re-login. However, if BaseApps are retired, attached clients will be migrated to another running BaseApp.

5.3.4. Secondary Databases

To reduce the load on the primary database and DBMgr, the BaseApp stores persistent data temporarily in a secondary database on its local disk. While an entity is active, changes to the entity's persistent data is stored only in the secondary database. When the entity is unloaded, its persistent data will be transferred from the secondary database to the primary database.

For details on secondary databases, see the document Server Programming Guide, chapter “Secondary Databases”.

5.4. BaseAppMgr

5.4.1. Implementation

When a BaseApp starts up, it registers itself with the BaseAppMgr. The BaseAppMgr maintains a list of all BaseApps.

5.4.2. Logging In

When a client logs in, the LoginApp makes a request to the BaseAppMgr (via the DBMgr) to add a player to the system. The BaseAppMgr then adds a proxy to the least loaded BaseApp. The BaseAppMgr sends the IP address of the proxy to the LoginApp. This is sent to the client, which then uses it for all future communication with the server.

5.4.3. Fault Tolerance

In case of failure, a Reviver restarts the BaseAppMgr. The BaseApps re-register with the BaseAppMgr, allowing it to continue normally.

For more details, see “Reviver” on page 39 .

5.5. LoginApp

5.5.1. Implementation

Each LoginApp listens to a fixed (configurable) port for requests to log in from clients.

The steps for client login are described in “Logging In” on page 17 .

All requests made by the LoginApp are non-blocking, so that it can handle many simultaneous logins.

5.5.2. Multiple LoginApps

In terms of load, a single LoginApp should be sufficient for most purposes. However, as this is still a potential bottleneck, and single point of failure, BigWorld supports running multiple LoginApps.

The remaining issue then is how to distribute the client login requests across multiple login servers. The standard way to do this is use a DNS solution similar to how popular web sites balance traffic load across multiple machines.

5.6. DBMgr

The DBMgr is the interface to the primary database. There are currently two implementations of the database interface component:

- XML
- MySQL

Others can also be added.

5.6.1. XML

The XML implementation saves data in a single XML file — `<res>/scripts/db.xml`. This is good for running small servers, since it is lightweight, and does not have any dependencies on an external database.

In this implementation, DBMgr reads the XML file on startup, and caches all player descriptions in memory. No disk I/O is performed until shutdown, when the entire XML file is written to disk.

The XML database has several known limitations that make it unsuitable for production or serious development environments. It differs significantly to the MySQL database in the area of login processing. For more details, see the document Server Programming Guide's section *User Authentication and Billing System Integration*.

5.6.2. MySQL

The MySQL implementation communicates with a MySQL database, via the native MySQL interface. MySQL can be running on the same machine, or a different machine, since the protocol works over TCP.

Each class of entity is stored in a separate SQL table, and each property of a class is a column. The tables are indexed by DatabaseIDs.

On startup, the SQL database schema is checked against the XML entity definition files (named `<res>/scripts/entity_defs/<entity>.def` — for details, see the document Server Programming Guide's section *Physical Entity Structure for Scripting* → "The Entity Definition File"), to ensure that they are consistent. If any entity class does not have a database table, then one is automatically created. If any class has new properties in the XML file that are not in the database, then columns are automatically added to the appropriate tables. This functionality is mainly useful during development, when the schema may change frequently.

5.6.3. Fault Tolerance

In case of a failure, the DBMgr can be restarted by a Reviver[2], which may be on a different physical machine. Once the DBMgr is restarted, it advertises its presence via BWMachined, and then all interested parties will start communicating with it.

The XML database is not fault-tolerant, since all data is held in RAM until shutdown. Although the DBMgr may be restarted by a Reviver, all updates since the server was last shutdown will be lost.

The DBMgr also plays an important role in BigWorld's second level of fault tolerance. For details, see the document Server Programming Guide, chapter *Disaster Recovery*.

5.7. Reviver

The Reviver is a watchdog process used to restart other processes that fail (either because the machine the process was running on has failed, or the process itself has crashed or is unresponsive). The Reviver is started on machines reserved for fault tolerance purposes, and waits for processes to attach themselves to it.

[2] For details, see "Reviver" on page 39 .

The processes that use Reviver include:

- CellAppMgr
- BaseAppMgr
- DBMgr
- LoginApp

When each of these processes is started, it searches the network for a Reviver, then attaches itself — if it is supported by Reviver (this configuration is done via `/etc/bwmachined.conf` — for details, see the document Server Operations Guide's section *Fault Tolerance*, "Fault Tolerance with Reviver", "Specifying Components to Support".).

The Reviver will then ping this process regularly, to check if it is available — if it fails to reply, the Reviver will restart itself as the failed process, replacing the process and the machine. The faulty machine/process can then be shut down for service.

Since Revivers normally stop running after reviving a process, generally a few of them are needed (preferably on different machines). The Revivers use BWMachined to start the new processes.

If a Reviver crashes, then the process requesting the monitoring will detect the missing pings and look for a new Reviver. It is up to the operator to ensure that enough Revivers are running on appropriate machines.

Even though it is not recommended, Reviver can also be started via the command line. For more details, see the document Server Operations Guide's section *Fault Tolerance*, "Fault Tolerance with Reviver", "Command-Line Options".

5.8. BWMachined

The BWMachined daemon runs on every machine in the server cluster, and is started automatically at boot time. Its responsibilities are:

- Start and stop server components.
- Locate server components.
- Provide machine statistics (CPU speeds, network stats, etc…).
- Provide process statistics (memory and CPU usage).

The following sub-sections describe each responsibility.

5.8.1. Start and Stop Server Components

BWMachined can be used to remotely start and stop server components. This functionality is used by the Server Control Tool.

It is possible to start components using any user ID, as well as to specify which build configuration should be used (**Debug**, **Hybrid**, or **Release**).

Since not all users will have their own build of the server, it is possible to specify the location of the server binaries to be used, on a per-user basis. This can be done either in the file `$HOME/.bwmachined.conf`, or in the file `/etc/bwmachined.conf`.

The file `/etc/bwmachined.conf` can also be used to specify the server components that should be supported by the Reviver. For details, see the document Server Operations Guide's section *Fault Tolerance*, "Fault Tolerance with Reviver", "Specifying Components to Support".

5.8.2. Locate Server Components

Whenever a server component starts, it can choose to publicise its Mercury interface. If it does, Mercury will send a registration packet to BWMachined on the local machine.

The list below describes the fields included in the packet:

- **`UID`** — User ID.
- **`PID`** — Process ID.
- **`Name`** — Name (*e.g.*, CellAppMgr)
- **`ID`** — An optional unique ID (*e.g.*, Cell ID).

BWMachined will then add the process to its list of registered processes. It polls the `/proc` file system every second, to determine CPU and memory usage for each process.

When a publicised server component shuts down, it must send a deregistration packet to the local BWMachined. If a process dies without deregistering, BWMachined will find this out when it next polls and updates its list of known components.

It is possible to search for server components that match certain fields by sending a query packet to BWMachined. It will reply by sending a response packet for each matching process.

For example, when a CellApp starts, it needs to find the address of the CellAppMgr process running under the current UID. It sends a broadcast query to every BWMachined on the network, asking for all processes that match the current UID, and the name "CellAppMgr".

5.8.3. Provide Machine Statistics

On startup, BWMachined examines the file `/proc/cpuinfo` to determine the number of CPUs and their speed. It also regularly monitors CPU, memory, and network usage for the whole machine. This information can be obtained via a UDP request, and is displayed in the StatGrapher component of WebConsole.

5.8.4. Provide Process Statistics

BWMachined polls the `/proc` file system every second, and gathers memory and CPU statistics for all registered processes. This information can be obtained via a UDP request, and is displayed in the StatGrapher component of WebConsole.

5.8.5. BWMachined Interface Discovery

In order to assist with error detection of misconfigured default broadcast addresses and reduce the amount of configuration required, BWMachined determines at startup which interface is configured as the default broadcast route[3].

If no `<internalInterface>` tag is defined in the `bw.xml` configuration file[4], then the server components will query BWMachined over the `localhost` interface, and request the interface to use as the internal interface.

If the `<monitoringInterface>` tag[5] is left undefined in `bw.xml`, then MessageLogger[6] will query BWMachined for the internal interface, and any server components will fall back to using the internal interface they have already discovered during startup for the monitoring interface.

---

[3] For details on how to setup the default broadcast route, see the Server Installation Guide's chapter *Cluster Configuration*, section "Routing".

[4] For details, see the Server Operations Guide's chapter *Server Configuration with `bw.xml`*.

[5] For details, see the Server Operations Guide's chapter *Server Configuration with `bw.xml`* → "General Configuration Options".

[6] For details, see the Server Operations Guide's chapter *Cluster Administration Tools* → "Message Logger".

Note

The `<internalInterface>` tags are deprecated, and while their behaviour is still consistent with previous releases, it is recommended to not use them.

Chapter 6. Other Features

6.1. IDs

An ID broker or issuing service is required, whenever the CellApp creates a new entity, it needs a unique entity ID. The CellAppMgr currently provides this functionality.

6.1.1. ID Allocation

The CellAppMgr currently acts as a central ID broker for IDs. It handles requests for a block of IDs, and allows unused IDs to be returned for 'recycling'.

If the CellAppMgr is restarted by a Reviver after crashing, then it will recover its state by querying the cells for the highest ID they are currently using. The sequence will be restarted from the highest ID of all, added by one. A small number of IDs may be wasted in the event of a failure.

Excess unused IDs are stored in special tables `bigworldNewID` and `bigworldUsedIDs`. For details, see the document Server Programming Guide's section *MySQL Database Schema* → "Non-Entity Tables".

6.2. Inter-Process Communication (Mercury)

6.2.1. Overview

Mercury is the network layer used for communications between the client and server, and between all server components. It is based on UDP, and allows both reliable and unreliable communications.

6.2.2. Nub

The Nub is the core of Mercury. It is responsible for sending and receiving packets, delivering timer messages, and general socket notifications.

Almost all communication uses UDP, a single socket is used for most connections.

The Nub normally controls the event loop of the application. So it also provides a time queue, and user socket notifications. The time queue can be used to invoke callbacks at a regular interval, or to invoke a single callback after a timeout, if required. User sockets can be monitored, and callbacks invoked when they are ready for reading.

6.2.3. Messages

A message is the basic unit of communication. It consists of:

- **Message type** — 1 byte.
- **Message size** — 0-4 bytes.
- **Message data** — Variable.

The message size is not needed for fixed-length messages. The message data is simply a stream of bytes that is interpreted by the client and server. Streaming operators are provided for most data types, to make marshalling easy.

6.2.4. Requests

A request is a message that expects an answer. It is possible to issue a request to a remote process, and receive a reply associated with it without blocking the process' execution. Internally, Mercury will allocate a unique request ID that is used to associate the reply with the request. When a request is made, a reply handler is

constructed to handle the reply. It also handles timeouts, and will invoke a timeout callback if no reply is received within a specified time.

6.2.5. Bundles

A bundle is a collection of messages that are sent and received as a unit. Grouping multiple messages together in a single packet reduces the overhead of UDP headers. If a bundle exceeds the maximum packet size, it will be split into multiple packets, and reassembled when it is received.

The maximum size of a packet is chosen based on efficiency and is related to the MTU (Maximum Transmission Unit) of a network. Each packet in a bundle may optionally contain additional data, including sequence numbers, request IDs, and acknowledgement IDs.

6.2.6. Channels

Channels are used to provide reliable communication between two Nubs. A channel may have different characteristics, depending on whether it is a client/server channel or a server/server channel and between which components it connects.

The table below describes the characteristics of some channels.

Channel	Latency	Bandwidth	Loss
Client/server	High	Low	High
Server/server	Low	High	Low

Channel's characteristics

A channel's traits are specified when it is created, so that its reliability algorithms can be appropriately selected.

If any message in a packet is flagged as reliable, the entire packet is also treated as reliable. The sender gives it a sequence number, and stores it in a sliding window. The receiver acknowledges the packet by placing an ACK on the next outgoing packet. Channels that have regular bi-directional communication (e.g. 10Hz for client/server, 50Hz for CellApp/CellApp) do not send the ACK immediately while other channels without regular communication will send this immediately.

If the sender receives an out-of-order ACK, then it will assume that the previous packets were lost, and resend them. For low-latency connections, there is also a threshold in which packets will not be resent for a certain time after they were previously sent.

For low-bandwidth connections (client/server), all unreliable messages are stripped from the packet before it is resent, and, if possible, the packet is piggybacked on the next outgoing packet, in order to save bandwidth.

6.2.7. Interfaces

Incoming messages on a Nub are handled by a single interface. Since the message type is a single byte, there can be up to 256 single-byte messages in an interface (actually, slightly less, since Mercury reserves several messages for internal use and a lot more if single-byte messages are multiplexed to more messages).

Interfaces are normally defined using the macros in `src/lib/network/interface_minder.hpp`. These are designed to hide the details involved in decoding the message and dispatching it. For simple fixed-length messages, the result is that a method is called on an object, with a structure containing the message parameters. For more complex variable-length messages, the method is called with a stream object, and parameters must be streamed off manually.

Interfaces also define whether each message is variable or fixed-length. For fixed-length messages, the size need not be sent, since the receiver knows how many bytes to expect.

6.3. Fault Tolerance and Disaster Recovery

Each component of the server has been designed to be fault-tolerant. Any of the server processes or machines can die unexpectedly, and the server should continue with little impact. See the description of server components for details about how each of them handles fault tolerance.

The BigWorld server also offers a second level of fault tolerance, known as disaster recovery. The server's state can periodically be written to the database. In the event of entire server failure, the server can be restarted using this information.

6.4. Packed Files

XML files are used throughout BigWorld due to its flexibility and ease of use. However, XML files are also widely regarded as being bloated and, when used in a computer game, too easily modified by the end user. Packed files are XML files converted to a more compact binary format. The client and all server components are able to load packed files where it usually expects an XML file. Replacing XML files with packed files will improve performance and offer a degree of obfuscation that should deter casual users from modifying the contents of the files.

Packed files are read-only and are usually unsuitable for use during development. Most of the tools (both client and server) will not work with packed files due to its read-only nature.