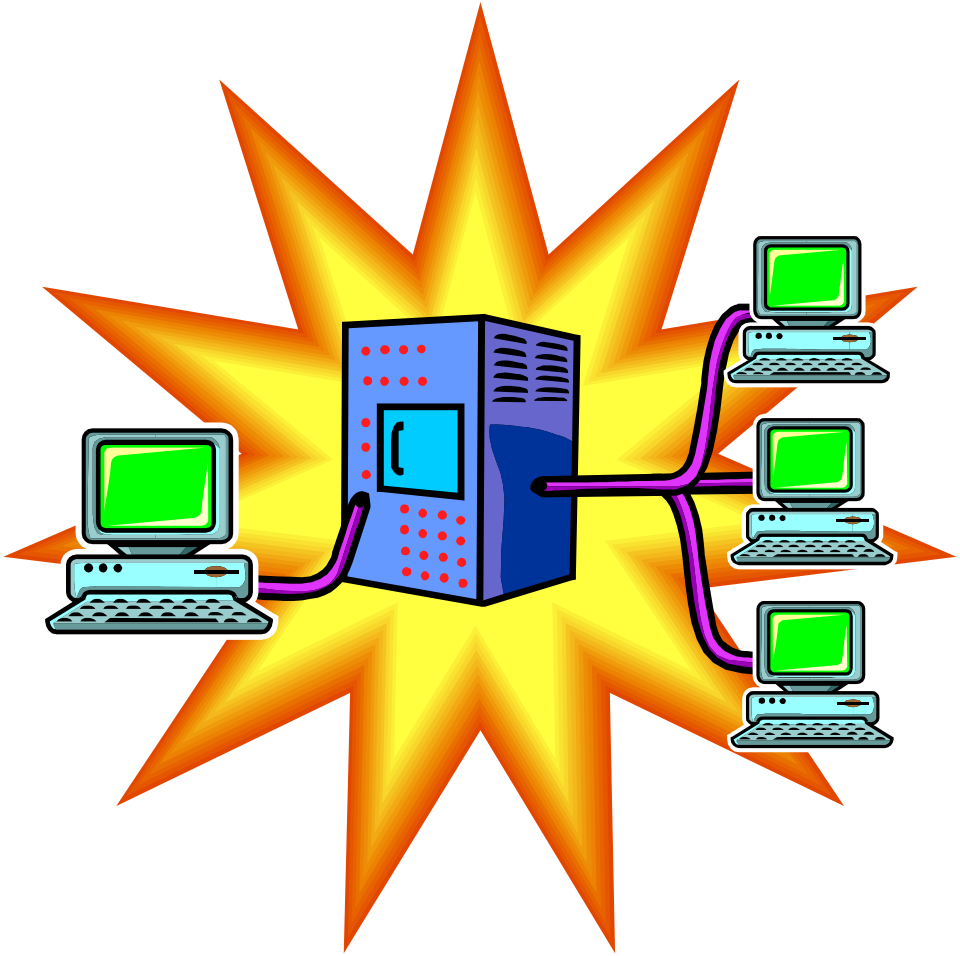
BigWorld Technology Training

General Concepts

Outline

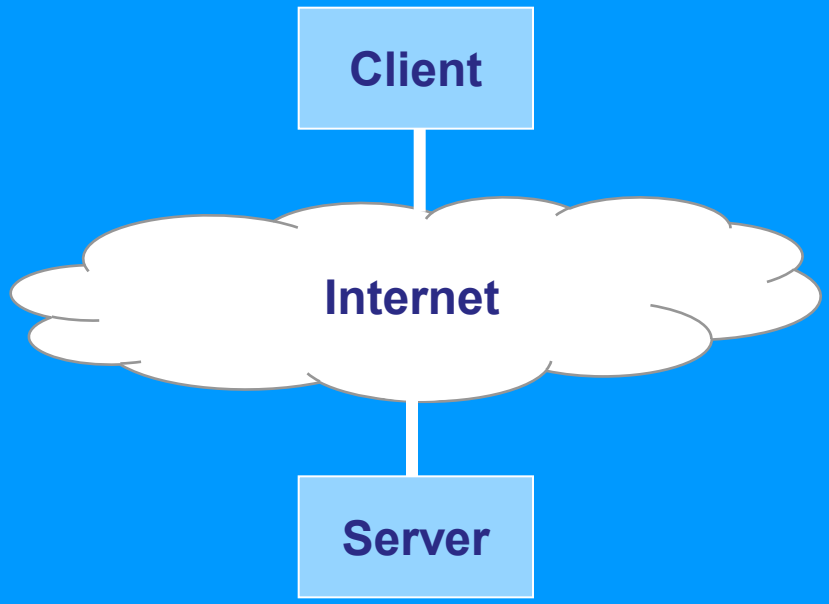
- System Architecture Overview
- Entities
- Spaces
- Game Resources
- Python API overview

Session 1
System Architecture Overview / Entities

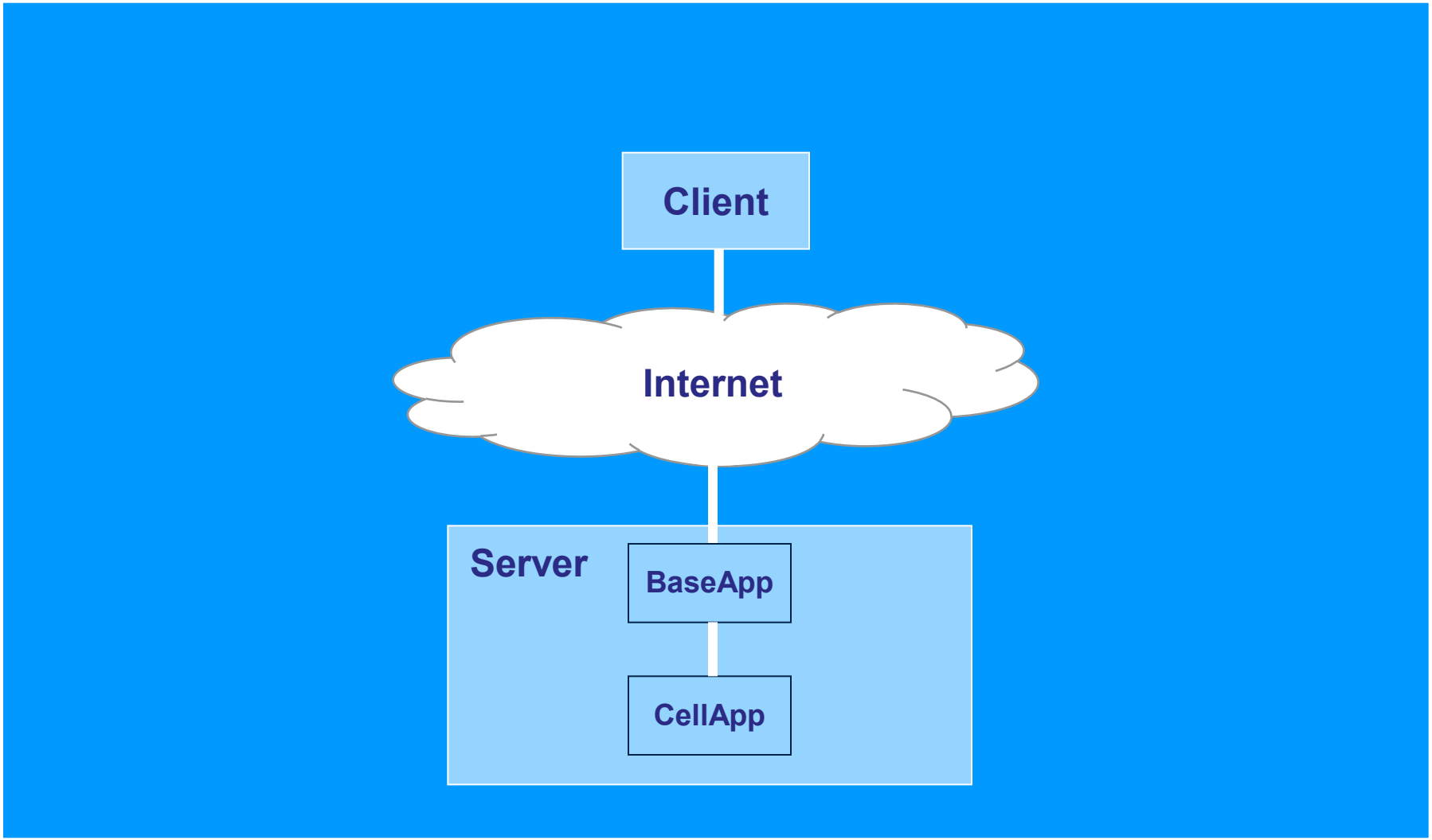


bigWORLD™ TECHNOLOGY

System Architecture Overview



System Architecture Overview



Server Architecture Overview

- Client
- BaseApp
  - Fixed communication point for the client
  - Proxies communication to and from CellApps
- CellApp
  - World process
  - Load balancing mechanism for the world environment based on number of active entities in the world

Entities

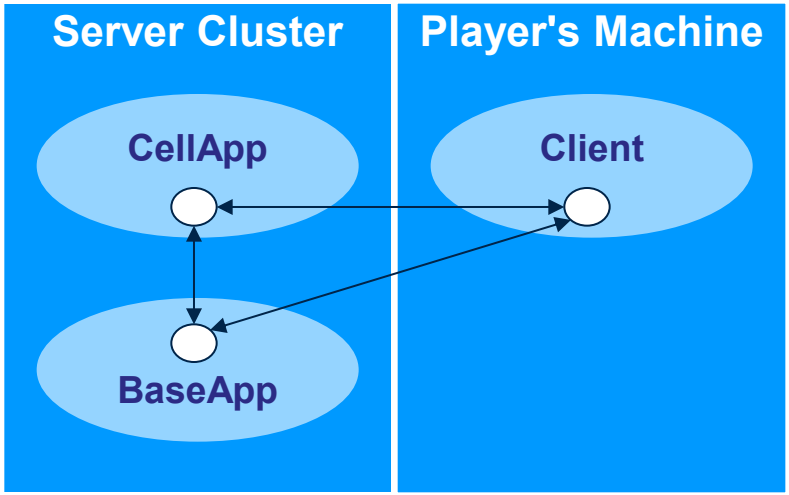
- *Entities* are game objects

  "Something that can be interacted with"

- Examples:
  - Players
  - NPC
  - Door
  - Seat

Entities

- Have a game wide Unique ID
- Implemented in Python
- Distributed objects
- Up to 3 different parts/contexts
  - Client
  - BaseApp
  - CellApp



Client Entity

- Implements visualisation for player
  - Particle effects applied to nearby players
  - Loading and switching models
  - Animation of models
- Stores data required to interact with visualised entities
  - Items in current player's bags
  - Currently equipped weapon
  - Player's current target

Cell Entity

- Similar to Client but focused on interaction rather than visualisation
- Stores data that allows other nearby entities to know about us
  - Health of an NPC
  - Currently equipped weapon
    - Used to determine damage caused by an attack

Base Entity

- No spatial knowledge
  - Account
  - Guild Manager
  - Auction House
- Longer term entity knowledge
  - Friends list of a player
- Used to persist across server restarts

Entity Properties

- Represent entity state in the current context
  - Weapon in left hand
  - Friends list
- Can exist in multiple contexts
  - Example: Player health (on Client and Cell)

Entity Properties

- Automatically propagated when modified
  - Controlled by property propagation flags
  - Client callback `set_<property_name>()`
- Propagation works with AoI
  - Only send property updates nearby entities
  - Priority of updates diminish with distance

Entity Methods

- Used to interact with an entity
  - Entity performing operations on itself
  - Entities interacting with each other
  - Performing operations between contexts
- Methods are associated with a single context
  - Example:
    - Base Entity implements `addToFriendsList( newFriend )`
    - Cell Entity implements `notifyNewFriend( newFriend )`
    - Client Entity implements `receiveFriendNotifaction( player,newFriend )`

Entity Methods

- Method calls on other entities are asynchronous
  - No return values (must use callbacks)
  - Only synchronous when entity is on the same component process
    - Don’t rely on this behaviour, it will cause scaling problems as you approach production.

Entity Methods

- Cell methods only called on *Real* entity
  - More details in the Server slides
- From the cell, client methods can be called on all clients within the AoI of an entity
  - Example:
    - Player yelling into the world "I am not an animal!"
    - All nearby players receive the text to display

Entity Definition

- Every entity must have a definition file
  - Commonly referred to as ‘entity def’ or ‘.def’
  - `<Entity_name>.def`
- Provides the same role as a C++ header or Java interface
- XML file
- Each entity type must exist in `entities.xml`

Entity Definition

- Defines:
  - Properties: where they exist and how to propagate
  - Methods: what methods are implemented in each context
- Can inherit from other entity definitions
- For a client to connect to the server all entity definition files must match

Entity Definition

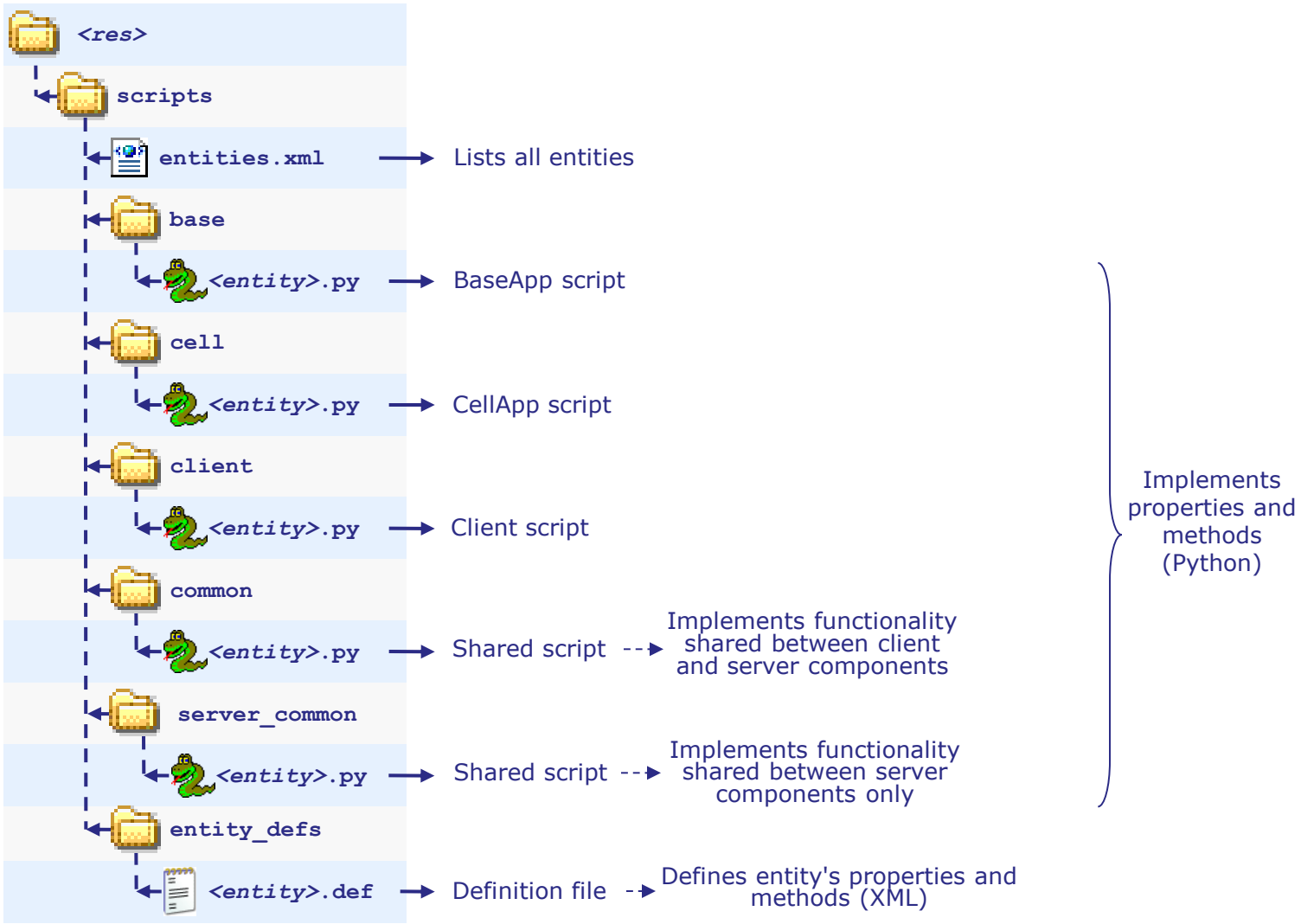
- Example:

```
<root>
   <Properties>
      <name>
         <Type>         STRING        </Type>
         <Flags>        ALL_CLIENTS   </Flags>
         <Persistent>   true          </Persistent>
      </name>
   </Properties>

   <ClientMethods>
   </ClientMethods>

   <BaseMethods>
   </BaseMethods>

   <CellMethods>
      <setName>
         <Exposed/>
      </setName>
   </CellMethods>
</root>
```

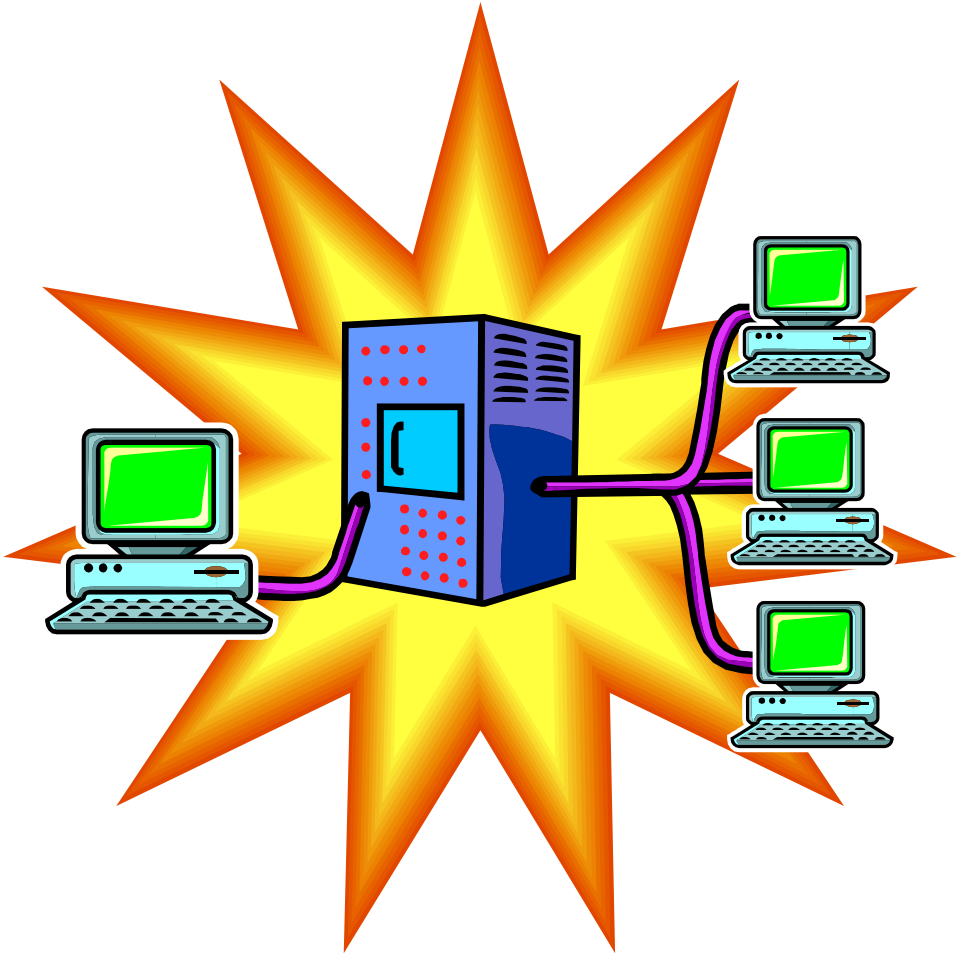
Entity Implementation

- Casing is important
  - Entity names used have to be consistent
  - .def must match .py implementation class

Entity Files



Session 2
Spaces

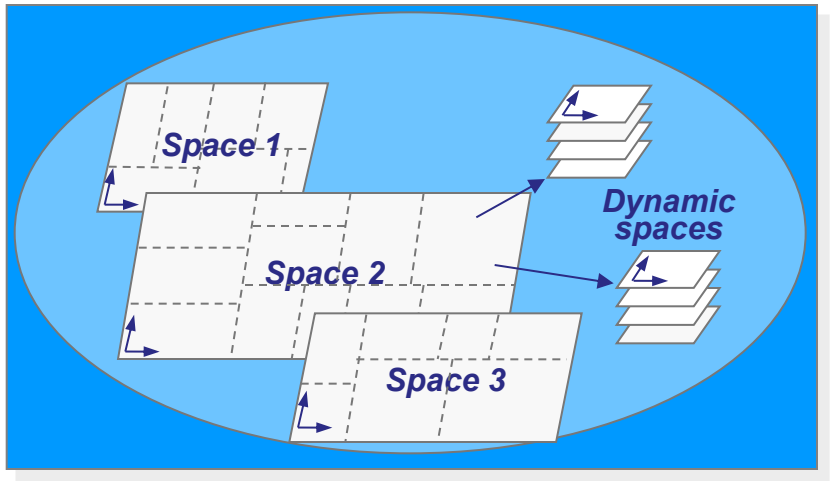


Spaces

- A space is the coordinate system a player views and interacts in.
- Can have multiple spaces (e.g. planets)
- Consists of:
  - Terrain
  - Static Models (houses, fountains, …)
  - NPCs (spawn locations)
  - NPC navigation information
  - Spatial metadata (User Data Object – details later)
- These can be shared between spaces.

Spaces

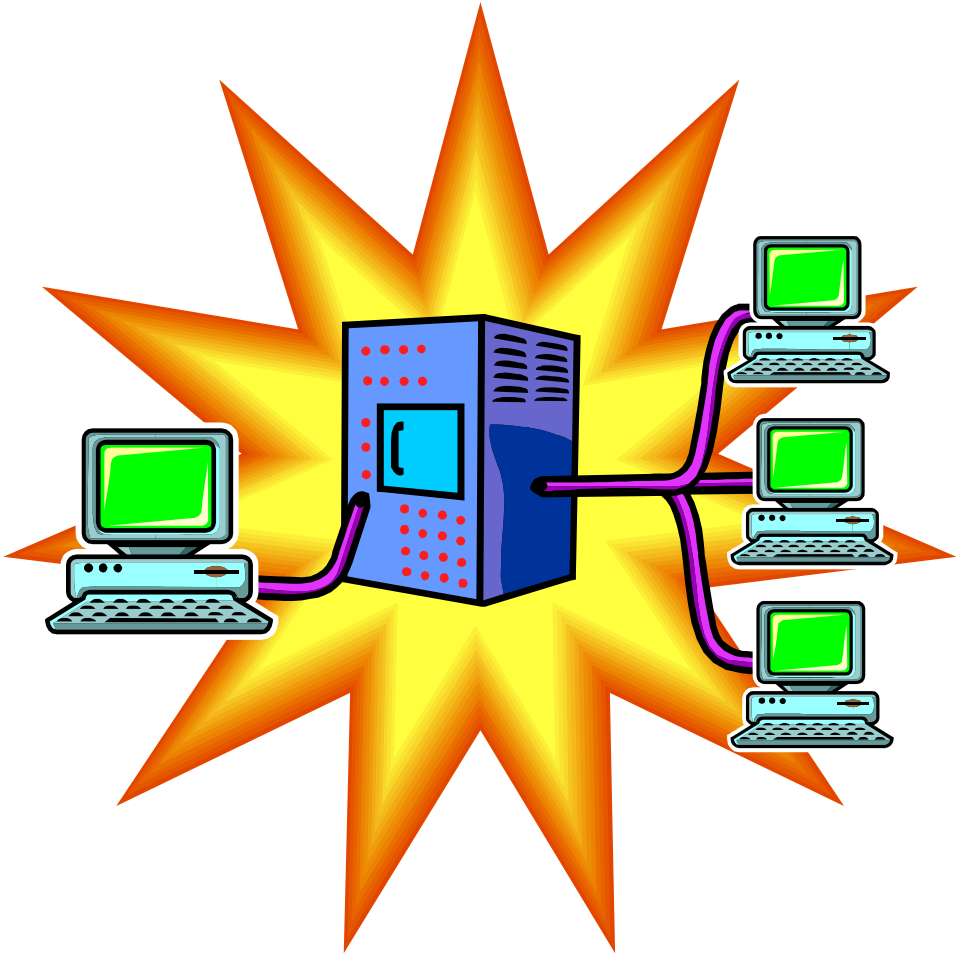
- Different kinds of space usage:
  - Multiple independent large spaces per server
    - Main game world that all players inhabit
  - Copies of a single space for isolated user experience
    - Instanced dungeon
    - Player's apartment



Space Geometry - Chunks

- Split up into chunks
  - 100m x 100m columns, x-z aligned
  - Unit of loading
- Chunks are loaded / unloaded as required
  - Client: loads chunks slightly beyond far plane
  - Server: loads area + 800m boundary of space being managed
  - Less aggressive unloading to avoid re-loading recently seen chunks

Session 3
Game Resources



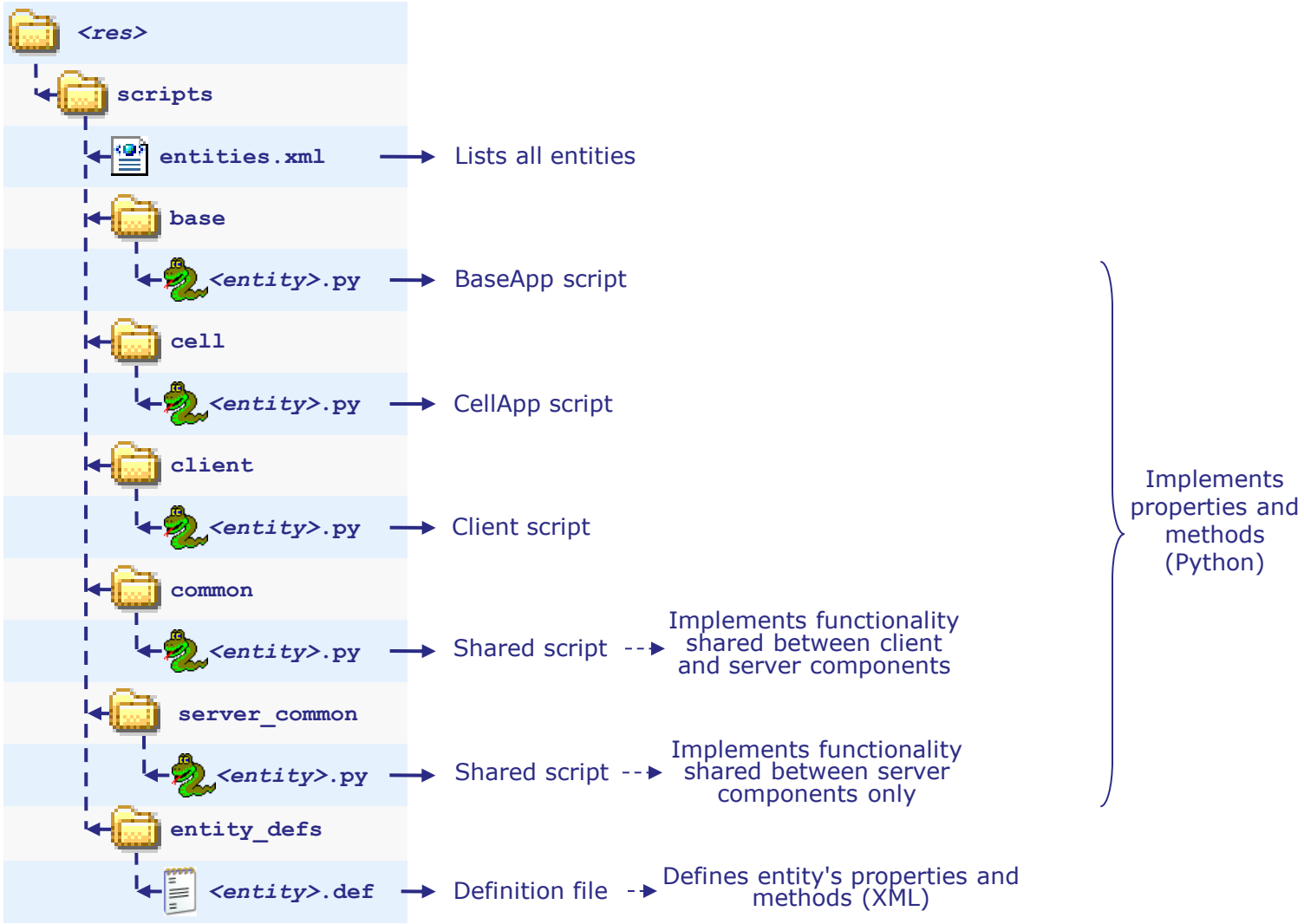
Game Resources

- Each game needs some resources in order to look and behave differently
  - Audio
  - Models
  - Animations
  - Textures
  - Code / Script
  - Space data

Game Resources

- BigWorld gains access to this data by defining `res` directories.
- Resource directories are the top level directory when accessing files in game.
- Each game project can incorporate multiple `res` directories
  - `<install_dir>/fantasydemo/res`
  - `<install_dir>/bigworld/res`

Game Resources



Game Resources

- Multiple resource directories overlay each other
  - Ordering of `res` directories is important
  - Example:
    - `/tmp/bw/fantasydemo/res`
    - `/tmp/bw/bigworld/res`
    - Both have a file `server/test.xml`
    - BigWorld loads: `/tmp/bw/fantasydemo/res/server/test.xml`

Game Resources

- Resource Manager (ResMgr)
  - Library to take the hassle out of dealing with resource hierarchies
  - Can read from different res directory types
    - Standard directories
    - Packed data (zip files)
  - Implemented as C++ library
  - Python module wrapping C++ functionality

Game Resources

- Files accessed with ResMgr are returned as a DataSection object
- DataSection
  - Handle to data
  - Data can be a single piece of information
    - Eg: A string “hello world”
  - Or a hierarchy of other DataSections
    - Eg: An XML document. Each child node is exposed as a DataSection

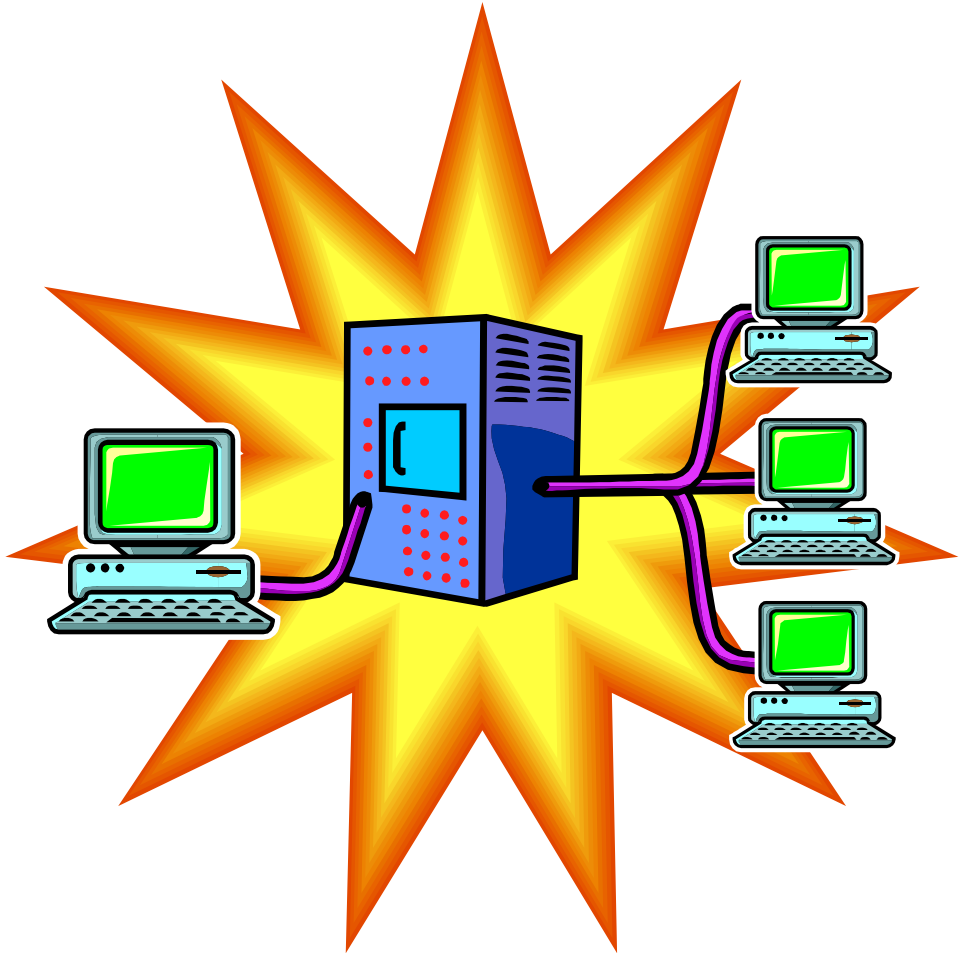
Asynchronous Loading

- When the game is running, it is important to load resources in a non-blocking way
- On the client, loading resources synchronously produces bad frame rates and blocks network processing
- On the server, loading resources synchronously blocks network processing and can cause process timeouts
- The client and the BaseApp each have a loading thread
- Loading resources synchronously at game initialisation is OK
- Importing Python modules after game initialisation is not recommended for the same reason

Asynchronous Loading

- On the client, implement `Entity.prerequisites()` and/or use `BigWorld.loadResourceListBG()`.
- On the BaseApp, use `BigWorld.fetchEntitiesFromChunks()`, `BigWorld.fetchFromChunks()`, or `BigWorld.fetchDataSection()`. User callback is called on completion.
- Warnings are issued if files are loaded in the main thread

Session 4
BigWorld Python API



Python API Overview

- Client / Server expose core component functionality via Python modules
- Modules with the same name on different components can implement different functionality

Client	Server	
	BaseApp	CellApp
BigWorld	BigWorld	BigWorld
GUI		
Math	Math	Math
Pixie		
ResMgr	ResMgr	ResMgr
<BWPersonality>	<BWPersonality>	<BWPersonality>

Python API Overview

- Simple client example:

```
import BigWorld

…

class Avatar( BigWorld.Entity ):

  def onEnterWorld( self, prereqs ):

    …

    deathWarp = BigWorld.Model( "models/fx/deathWarp.model" )

    self.addModel( deathWarp )
```

Python API – Client Modules

- BigWorld
  - World related client interface
  - Many sub-modules to handle
    - Animation
    - Entity interaction
    - Movement physics
    - ..and much more

Python API – Client Modules

- <BWPersonality>
  - Imported automatically
  - Personality name defined in game configuration file
  - Contains callback methods for events:
    - Interface (window resizing)
    - Input (key / mouse)
    - Game time change

Python API – Client Modules

- GUI
  - Used to create game interface (menus..etc)
  - Handles input events passed from personality script and delegates to subcomponents
- Math
  - Interface to C++ implementations of:
    - Vector (2/3/4)
    - Matricies
    - … and specialisations of these

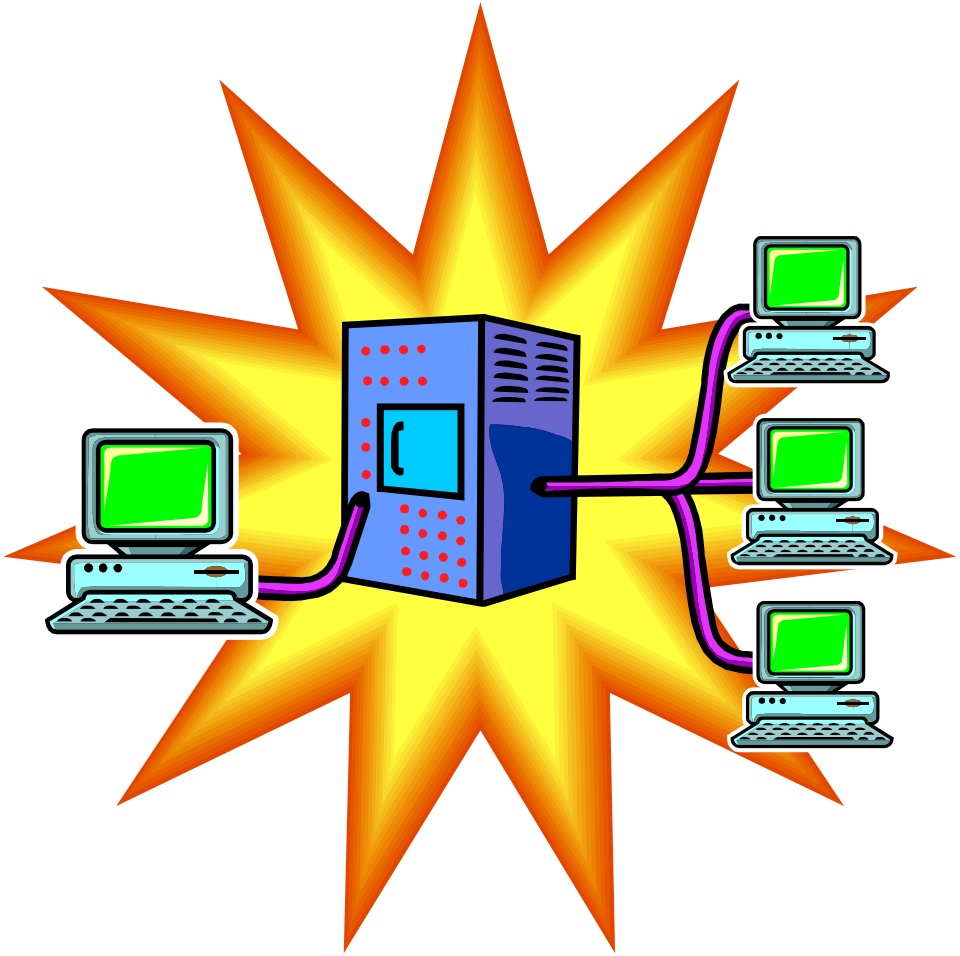
Python API – Client Modules

- Pixie
  - Interface to BigWorld particle systems
  - Used for creating short term particle effects
    - Flare off an engine
    - Lightning from a player spell
- ResMgr
  - Interface to access game resources
  - Read / Write / Delete / Exists

Python API – Server Modules

- Math / ResMgr
  - Effectively the same as client
- <BWPersonality>
  - Server process / event notification
- BigWorld
  - CellApp
    - Entity / World based server side interaction
  - BaseApp
    - Game-wide entity creation
    - Player account management / interaction

Session 5
How To Work Efficiently



Working Efficiently

- Developing a game requires many people working together
- Use source control
  - SVN / Perforce… etc
- Artists
  - Use bwlockd to co-ordinate World Building

Working Efficiently

- Windows users mount resources on server machines
  - Each developer (and even artists) should have their own account on a Linux server for testing
  - Share your game directories from Windows
  - Mount the drives into your user account
  - Start a server to test changes
- Run an office wide server that is kept up and running as a quick test point
  - Especially useful for artists

Working Efficiently

- Entity script development
  - Use the client Python console (DEBUG-P)
  - Connect to Python console on a live server (more in Server training)
- Plan to scale!
  - Take into account the size you want your game to be
  - Plan ahead how to achieve this
  - Test scaling as you progress through milestones