How To Implement Items And Trading

BigWorld Technology 1.9.1. Released 2008.

Software designed and built in Australia by BigWorld.

Level 3, 431 Glebe Point Road
Glebe NSW 2037, Australia
www.bigworldtech.com

Table of Contents

Chapter 1. Introduction

This document describes one possible solution for the implementation of items and inventory systems using BigWorld Technology. It also presents an example in which the back end of a system allows two players to securely trade items between themselves. It illustrates just one of the many ways in which atomic trades can be implemented.

Chapter 2. Items

Items are things that players and other entities can possess, such as guns, swords, loot, etc.

Since these are, most of the time, attached to other entities and do not have a position of their own in the world, they do not need to be BigWorld entities. The only exceptions are items lying on the ground. For more details, see "Dropping and picking items up" on page 10 .

2.1. Item definition

The simplest way of representing an item is as an integer value, representing the *item type ID*. This *type ID* can then be transmitted between the server and clients.

It is recommended that an *alias* be defined for the item type in the file `<res>/scripts/entity_defs/alias.xml` (where `<res>` is the first folder specified in environment variables `BW_RES_PATH`). In the sample implementation, the definition looks like this:

```
<root>
    ...
    <ITEMTYPE>    INT32    </ITEMTYPE>
    ....
```

The items types are enumerated inside the file `<res>/scripts/common/ItemBase.py`:

```
class ItemBase:
    ...
    NONE_TYPE           = -1
    STAFF_TYPE          =  2
    STAFF_TYPE_2        =  3
    DRUMSTICK_TYPE      =  4
    SPIDER_LEG_TYPE     =  5
    BINOCULARS_TYPE     =  6
    SWORD_TYPE          =  7
    SWORD_TYPE_2        =  9
    GOBLET_TYPE         =  17
    ...
```

Mapping from *item type ID* to specific item look and behaviour can be coded on the client and server as needed. This can be done by simply using lookup tables or item classes. In the sample implementation, item classes are used on the client to implement the behaviour of each type of item. All items types derive from a base Item class and must specialize items behaviours like `enact`, `enactIdle`, `enactDrawn` and `use`.

- **In `<res>/scripts/client/Item.py`:**

```
import ItemBase
....

class Item( ItemBase.ItemBase ):
    def use( self, user, target ):
        ...
    def name( self ):
        ...

class Food(Item):
    def use( self, user, target ):
        ....
```

```
        user.eat( self.itemType )
        user.cell.eat( self.itemType )
        ...
```

The type classes are also used to carry information about the items look, like models and icons.

- **In `<res>/scripts/client/Item.py`:**

```
class Food(Item):
    ...
    modelNames = {
        STRIFF_DRUMSTICK: "sets/items/item_food_drumstick.model",
        SPIDER_LEG:       "characters/npc/spider/spider_leg.model",
        WINE_GOBLET:      "sets/items/grail.model"
    }
    ...
    guiIconNames = {
        STRIFF_DRUMSTICK: "gui/maps/icon_items/icon_food_drumstick.tga",
        SPIDER_LEG:       "gui/maps/icon_items/icon_spider_leg.tga",
        WINE_GOBLET:      "gui/maps/icon_items/icon_grail.tga"
    }
    ...
```

Within this system, creating a new item type is done by inserting a new *item type ID* into the items type list and implementing class for the specialized behaviour.

You can build more complex structures to represent an item as needed, such as adding its ammo count. This information could be encoded into the `int32` data type, or a new class structure could be created to implement it.

2.2. Additional Security

The creation and deletion of items are best done using a *global items* manager on the base. This manager should ensure that items are created and destroyed according to specific rules, assign unique serial number to items, and enforce security so that things such as duplication exploits cannot be used.

2.3. Rendering the Item

For the item currently equipped by an Avatar to be properly rendered on all clients, the property containing the *item type ID* must be current on all clients at all times. Whenever a player chooses to equip a new item, it must notify its cell. The cell will then update all other clients with the newly selected *item type ID*.

In the sample implementation, the property `rightHand` carries the Avatar's currently equipped item. The fact that this property is flagged `OTHER_CLIENTS` means that the owner `Avatar` in the client will not be updated when the property changes in the cell (this is because changes to it are triggered by the client `Avatar` itself).

- **In `<res>/scripts/entity_defs/interfaces/Avatar.def`:**

```
<root>
    ...
    <Properties>
        <rightHand>
            <Type> ITEMTYPE </Type>
            <Flags> OTHER_CLIENTS </Flags>
            <Default> -1  </Default>
            <Editable> true </Editable>
```

```
    </rightHand>
    ...
```

- **In `<res>/scripts/client/Avatar.py`:**

```
class PlayerAvatar:
    ...
    def equip( self, itemType ... ):
        ...
        self.rightHand = itemType          # change locally
        self.cell.setRightHand( itemType ) # tell the world
        ...
```

- **In `<res>/scripts/cell/Avatar.py`:**

```
def setRightHand( self, sourceID, itemType ):
    ...
    self.rightHand = itemType # will propagate to all otherClients
```

To render the item in the player's hand, the 3D model representing the item must be attached to a hard point in the Avatar model. This should be done whenever the player equips a new item. The sample implementation uses the `set_rightHand` method to do that. The `set_rightHand` method is implicitly called whenever the value of the property `rightHand` is updated in the client by the server.

- **In `<res>/scripts/client/Avatar.py`:**

```
class Avatar:
    def set_rightHand( self, oldRH = None ... ):
        ...
        self.lockRightHandModel( True )
        ...

    def lockRightHandModel( self, lock, itemLoader = None ):
        ...
            self.rightHandItem = Item.newItem( self.rightHand )
            ...
            if self.rightHandItem != None:
                ...
                self.model.right_hand = self.rightHandItem.model
                ...
        ...
```

2.4. The Inventory

An inventory is a collection of items. Inventories are typically stored on both the client and the base. Having the inventory on the cell would increase the load of migrating entities from cell to cell as they move in the space.

The authoritative copy of an inventory should always reside on the base entity, where it can be easily stored and retrieved from persistent storage through the BigWorld database interface. The client copy of the inventory can then be regenerated from the base whenever necessary.

The client should have a graphics user interface to the inventory, so that players can view their items. They can use this interface to add or remove items from their inventories. This should then be updated on the base entity and stored on the database at convenient times.

In the sample implementation, the inventory data is stored in a set of `BASE_AND_CLIENT` properties in the Avatar entity:

- `inventoryItems`
- `inventoryLocks`
- `inventoryGoldPieces`

`inventoryItems` is an array of the type `InventoryEntry`. `inventoryLocks` is an array of the type `LockedEntry`. Both `InventoryEntry` and `LockedEntry` types are defined in the *<res>*`/scripts/entity_defs/alias.xml file`. These types are defined using BigWorld's `FIXED_DICT` type feature. `inventoryGoldPieces` is of type `INT32`.

- **In *<res>*`/scripts/entity_defs/Avatar.def`:**

```
<root>
    <Properties>
        ...
        <inventoryItems>
            <Type>  ARRAY <of> InventoryEntry </of> </Type>
            <Flags  BASE_AND_CLIENT </Flags>
            <Persistent> true   </Persistent>
            <Default>  ...   </ Default >
        </inventoryItems>

         <inventoryLocks>
             <Type>   ARRAY <of> LockedEntry </of> </Type>
             <Flags> BASE_AND_CLIENT </Flags>
             <Persistent> true   </Persistent>
        </inventoryLocks>

        <inventoryGoldPieces>
            <Type>  INT32   </Type>
            <Flags>  BASE_AND_CLIENT </Flags>
            <Persistent> true   </Persistent>
            <Default> 100 </Default>
        </inventoryGoldPieces>
        ....
```

- **In *<res>*`/scripts/entity_defs/alias.xml`:**

```
<root>
    ...
    <ITEMSERIAL>  INT32  </ITEMSERIAL>
    <LOCKHANDLE>  INT32  </LOCKHANDLE>
    <GOLDPIECES>  INT16  </GOLDPIECES>

    <InventoryEntry> FIXED_DICT
        <Properties>
            <itemType>
                <Type> ITEMTYPE </Type>
            </itemType>
            <serial>
                <Type> ITEMSERIAL </Type>
            </serial>
            <lockHandle>
                <Type> LOCKHANDLE </Type>
            </lockHandle>
       </Properties>
```

```
    </InventoryEntry>

    <LockedEntry> FIXED_DICT
        <Properties>
            <lockHandle>
                <Type> LOCKHANDLE </Type>
            </lockHandle>
            <goldPieces>
                <Type> GOLDPIECES </Type>
            </goldPieces>
        </Properties>
    </LockedEntry>
    ....
```

The fact that the inventory properties are flagged `BASE_AND_CLIENT`, means that:

- When initialised, the owning client entity will be carrying an exact copy of the properties as set on the base entity
- The inventory property will not exist neither in the cell entity nor in client Avatars not controlled by the owning player.
- Changes to the inventory property's value in the base will not be propagated to the client. The game logic must take care of keeping both copies synchronized as changes are made to the inventory.

  One example of such logic is an item being added to the inventory after being picked up by the player:

  - **After the pick up, the base adds the item into the inventory and notifies the cell.**
  - **The cell forwards the notification to the client.**
    - **In `<res>/scripts/base/Avatar.py`:**

```
def pickUpResponse( self, success, droppedItemID, itemType ):
    ...
    itemsSerial = self.inventoryMgr.addItem( itemType )
    self.cell.pickUpResponse( True, droppedItemID, itemType, itemsSerial
 )
    ...
```

    - **In `<res>/scripts/cell/Avatar.py`:**

```
def pickUpResponse( self, success, droppedItemID, itemType, itemSerial ):
    ...
    self.client.pickUpResponse( True, droppedItemID, itemSerial )
    ...
```

  - **In the client, the Player replicates the addition of the item into the inventory.**
    - **In `<res>/scripts/client/Avatar.py`:**

```
class PlayerAvatar:
    ...
    def pickUpResponse( self, success, droppedItemID, itemSerial ):
        ...
        droppedItem = BigWorld.entities[ droppedItemID ]
        self._pickUpProcedure( droppedItem, itemSerial )
```

```
        ...

    def _pickUpProcedure( self, droppedItem, itemSerial ):
        ...
        itemType = droppedItem.classType
        self.inventoryMgr.addItem( itemType, itemSerial )
        ...
```

2.4.1. Inventory Manager

In the sample implementation, the class `InventoryMgr` encapsulates the logic of adding, removing, selecting, locking, and trading items in the inventory. The `InventoryMgr` is initialised with a reference to the inventory holder entity. It looks for the inventory properties (`inventoryItems`, `inventoryLocks` and `inventoryGoldPieces`) in that entity.

Only changes made to the entity's properties (on the base) will be written to the database. `InventoryMgr` member variables will not persist, and should be treated as temporary. One such variable is `_currentItemIndex`, which is not considered a persistent property and is initialised to `NOITEM` (-1) every time a new inventory is instantiated.

Adding and removing items to/from the inventory is just a matter of appending and popping items into/from the `inventoryItems` array, taking care of respecting locking rules, if any, and internal consistency constraints, like resetting the selected item, if it is the one being removed from inventory.

- **In `<res>/scripts/common/Inventory.py`:**

```
NOLOCK = -1
NOITEM = -1
...
def __init__( self, entity ):
    self._entity = weakref.proxy( entity )
    self._curItemIndex = NOITEM
    ...

def addItem( self, itemType ... ):
    ...
    if itemSerial is None:
        itemSerial = self._genItemSerial()
    entry = { "itemType": itemType, "serial": itemSerial, "lockHandle":
 NOLOCK }
    self._entity.inventoryItems.append( entry )
    ...
    return itemSerial

def removeItem( self, itemSerial ):
    index = self._itemSerial2Index( itemSerial ) # throws is serial not
 found
    entry = self._retrieveIfNotLocked( index )   # throws if item is locked
    try:
        item = inventory[ itemIndex ]
        inventory.pop( itemIndex )
    except IndexError:
        errorMsg = 'removeItem: invalid item index (idx=%d)'
        raise IndexError, errorMsg % itemIndex

    if self._curItemIndex == index:
        self._curItemIndex = -1
    elif self._curItemIndex > index:
        self._curItemIndex -= 1
    ...
```

```
    return entry[ "itemType" ]
```

Note

Instead of a straight reference, a `weakref` to the entity is kept to avoid creating a cyclic reference that will prevent the entity from being eventually deleted.

2.4.2. Serial Numbers

Note that in the sample implementation items are not referenced by their index inside the `inventoryItems` array. Instead, they are assigned a serial number when added to the inventory, and are referenced by that serial throughout all their life inside it.

The reason for this is to allow simultaneously changes to the inventory from multiple sources (*e.g.*, game client, web interface, mobile device). In this case, if direct indices are used, references to items can become obsolete while a request is still being processed.

In the example, serial numbers are not attached to the item itself, but to their existence in an inventory, that is, serial numbers are guaranteed to be unique only within a single inventory. Two inventories can contain items with serial numbers duplicated between them.

A more comprehensive items systems can use a global serial number generator and have them assigned to items when they are first created. This serial numbers can then be used throughout all game subsystems to uniquely refer to items, to track duplicated items, etc.

When using serial numbers it is important to have a single authoritative copy of the inventory generating the serials for each item, otherwise serial numbers can become inconsistent between instances of the same inventory. When adding an item to the non-authoritative copy of the inventory, its assigned serial number must be provided along with the item.

- **In `<res>/scripts/common/Inventory.py`:**

```
def addItem( self, itemType, itemSerial = None ):
    ...
    entry = { "itemType": itemType, "serial": itemSerial, "lockHandle":
 NOLOCK }
    self._entity.inventoryItems.append( entry )
    ...
    return itemSerial
```

2.5. Dropping and picking items up

As mentioned above, the items can usually be represented as a property of an *entity* (or an entry into an array property). However, there is one case where this is not sufficient, and the item needs to be an *entity*.

This is the case of an item that has been dropped on the ground. The reason for this is that if you walk away from an item that you dropped, other people still need to see it.

If the item was just a property of a player, then when the player left the *area of interest* (which is usually about 500m), the property would disappear as well, and so would the dropped item.

On the other hand, if the dropped item itself is an entity, then other players will see it, as long as it is in their *area of interest*.

Making a dropped item an entity means that you can write interaction scripts just like any other entity in the world. You can target it, shoot it, pick it up, etc...

The process of dropping and picking items up should be something similar to the steps described below.

2.5.1. Dropping an item on the ground

Initially, the item is in the player's inventory, expressed as an entry in the *inventoryItems* property, the `itemType` field in the entry holds the *item type ID*.

A game would follow the steps described below for an item drop:

1. Player uses the user interface to drop an item.

2. Because the item may be unavailable in the server (*e.g.*, in case it has just been locked in the inventory from a web interface to the trading system), the client Avatar makes the drop item requests to the base. If the player can drop the item, the base removes the item from the inventory and notifies the cell about the dropped item:

   - **In `<res>/scripts/client/Avatar.py`:**

     ```
     class PlayerAvatar:
         ...
         def dropOnline( self ):
             ...
             self.base.dropRequest( itemSerial )
     ```

   - **In `<res>/scripts/base/Avatar.py`:**

     ```
     def dropRequest( self, itemSerial ):
         ...
         itemType = self.inventoryMgr.removeItem( itemIndex )
         self.cell.dropNotify( True, itemType )
         ...
     ```

3. The cell creates a `DroppedItem` entity that matches the item dropped by the player:

   - **In `<res>/scripts/cell/Avatar.py`:**

     ```
     def dropNotify( self, success, itemType ):
         ...
         BigWorld.createEntity( "DroppedItem", ... )
         ...
         self.rightHand = ItemBase.ItemBase.NONE_TYPE
     ```

4. The server informs all clients within the *area of interest* that a `DroppedItem` has been created (via Python's function `Bigworld.createEntity`).

5. The confirmation to the Avatar for his drop request comes from the `DroppedItem` itself. The client then plays the drop animation and removes the item from player's right hand (for simplicity, the code that synchronizes the drop animations, the item model being removed from the avatar's right hand and reappearing on the ground is not shown here):

   - **In `<res>/scripts/client/DroppedItem.py`:**

     ```
     def enterWorld( self ):
         ...
         dropper = BigWorld.entities[ self.dropperID ]
     ```

```
    dropper.dropNotify( self )
    ...
```

- **In `<res>/scripts/client/Avatar.py`:**

```
class Avatar:
    ...
    def dropNotify( self, droppedItem ):
        ...
        self._dropProcedure( droppedItem )

    def _dropProcedure( self, droppedItem ):
        ...
        droppedItem.dropComplete()
        ...
```

6. All clients now draw the correct model for the `DroppedItem` entity on the ground:

- **In `<res>/scripts/client/DroppedItem.py`:**

```
class DroppedItem( BigWorld.Entity ):
    ...
    def dropComplete( self ):
        ...
        self._showModel()

    def _showModel ( self ):
        ...
        self.model = self.item.model
```

2.5.2. Picking an item up from the ground

Initially, the item is a `DroppedItem` entity lying on the ground, as described in “Dropping an item on the ground” on page 11 .

A game would follow the steps described below for picking up an item:

- **In `<res>/scripts/client/Avatar.py`:**

```
class PlayerAvatar:
    ...
    def pickExecute( self, droppedItem ):
        ...
        self.cell.pickUpRequest( droppedItem.id )
        ...
```

- **In `<res>/scripts/cell/Avatar.py`:**

```
def pickUpRequest( self, sourceID, droppedItemID ):
    ...
    item = BigWorld.entities[ droppedItemID ]
    item.pickUpRequest( self.id )
    ...
```

- **In `<res>/scripts/cell/DroppedItem.py`:**

```
def pickUpRequest( self, whomID ):
    ...
    if self.pickerID == 0:
        picker = BigWorld.entities[ whomID ]
        picker.base.pickUpResponse( True, self.id, self.classType )
        self.addTimer( 5, 0, DroppedItem.DESTROY_TIMER )
        self.pickerID = whomID
    ...
```

- In **<*res*>/scripts/base/Avatar.py**:

```
def pickUpResponse( self, success, droppedItemID, itemType ):
    if success:
        itemsSerial = self.inventoryMgr.addItem( itemType )
        self.cell.pickUpResponse( True, droppedItemID, itemType, itemsSerial
 )
    ...
```

- In **<*res*>/scripts/cell/Avatar.py**:

```
def pickUpResponse( self, success, droppedItemID, itemType, itemSerial ):
    if success:
        # sucess:notify all clients this entity base
        self.client.pickUpResponse( True, droppedItemID, itemSerial )
        self.otherClients.pickUpNotify( droppedItemID )
        self.rightHand = itemType
    ...
```

- In **<*res*>/scripts/client/Avatar.py**:

```
class Avatar:
    ...
    def pickUpNotify( self, droppedItemID ):
        ...
        droppedItem = BigWorld.entities[ droppedItemID ]
        self._pickUpProcedure( droppedItem )
        ...

class PlayerAvatar
    ...
    def pickUpResponse( self, success, droppedItemID, itemSerial ):
        ...
        droppedItem = BigWorld.entities[ droppedItemId ]
        self._pickUpProcedure( droppedItem, itemSerial )
        ...
```

- In **<*res*>/scripts/client/Avatar.py**:

```
class PlayerAvatar:
    ...
    def pickUpFinish( self, droppedItem ):
        ...
        itemType  = droppedItem.classType
        self.inventoryMgr.addItem( itemType, itemSerial )
        self.inventoryMgr.selectItem( itemSerial )
```

```
        ...
```

- In **<*res*>/scripts/cell/DroppedItem.py**:

```
def pickUpRequest( self, whomID ):
    ...
    self.addTimer( 5, 0, DroppedItem.DESTROY_TIMER )

def onTimer( self, timerId, userId ):
    ...
    if ( userId == DroppedItem.DESTROY_TIMER ):
        self.destroy()
```

1. Client requests to pick item up.
2. The item locks itself as being picked up by the requesting Avatar. Further requests for pickup will be denied by the server. The item also notifies the requester base about the pickup so that the item is added to the player inventory. Finally, the item sets a timer to remove itself from the world.
3. The cell notifies all clients about the pick up and updates the avatar's right hand property.
4. All clients are notified that the Avatar is picking up the item. A picking item up animation is started on the Avatar's model. Note that, although other clients will see the item model in the player hands as a consequence of `rightHand` being set in the server, the `PlayerAvatar` will not have its `rightHand` property updated and must explicitly equip the item (remember that `rightHand` is flagged `OTHER_CLIENTS`).
5. The Player adds the picked item into his inventory and selects it.
6. On the base, the timer goes out and the `DroppedItem` entity destroys itself.
7. The item has been added to the player inventory in both the client and the server. All clients draw the item in player's right hand. The dropped item has been removed from the server.

2.6. Locking inventory items

If the game design calls for some sort of asynchronous transactions to be carried out with inventory items, it will be necessary to implement locking of items inside the inventory.

During asynchronous transactions, like those taking place over a mobile device or web interface, a player may offer one or more items for trading. A second player may, at his own time, inspect the items being offered and in turn reply the offer with one or more of his items. When the first player inspects and accepts the items in the reply offer, the transaction can be carried out.

Note

Although this is not a requirement for asynchronous transactions, locking also offers a way to reference a set of items (and maybe also some amount of currency) using a convenient single handle.

From first offer to final acceptance, a finite amount of time will have passed, from a couple of seconds to several hours or even days. Problems will occur if the items initially offered are no longer available when both parties accept the transaction (if, for example, one of the players drops, consumes or trades with a third player any of the offered items).

One approach to avoid the problem is to lock the items inside the inventory once they have been committed to a trade offer. Locked items should not be available for dropping, offering on a second trade nor equipping

(thus consuming). The player should be free to unlock the items at any time. Unlocking an item should invalidate the trade associated with it.

In the sample implementation, items are flagged as locked by assigning a valid lock handle to the `lockHandle` field in the item's entry in the `inventoryItems` array. Some amount of gold pieces can also be locked together with the set of items. The gold information is stored in the `inventoryLocks` array. Locking items and gold pieces yields a lock handle. This lock handle can than be used to reference the locked lot, either to unlock it or to trade it for gold, other items or a combination of both.

- **In `<res>/scripts/common/Inventory.py`:**

```
NOLOCK = -1
...
def itemsLock( self, itemsSerials, goldPieces ):
    lockHandle = self._getNextLockHandle()
    self.itemsRelock( lockHandle, itemsSerials, goldPieces )
    return lockHandle

def itemsRelock( self, lockHandle, itemsSerials, goldPieces ):
    ...
    itemsIndexes = []
    for serial in itemsSerials:
        index = self._itemSerial2Index( serial )
        if self._entity.inventoryItems[ index ][ "lockHandle" ] != NOLOCK:
            errorMsg = 'Item item already locked (idx=%d)'
            raise LockError, errorMsg % index
        itemsIndexes.append( index )

    for index in itemsIndexes:
        self._entity.inventoryItems[ index ][ "lockHandle" ] = lockHandle

    lockedEntry = { "lockHandle": lockHandle, "goldPieces": goldPieces }
    self._entity.inventoryLockedItems.append( lockedEntry )
...
def itemsUnlock( self, lockHandle ):
    index = self._getLockedItemsIndex( lockHandle )
    lockedEntry = self._entity.inventoryLockedItems[ index ]
    self._entity.inventoryLockedItems.pop( index )
    for entry in self._entity.inventoryItems:
        if entry[ "lockHandle" ] == lockHandle:
            entry[ "lockHandle" ] = NOLOCK
```

Chapter 3. Trading

This chapter describes one of the possible ways in which players can buy, sell, or exchange goods between themselves or with *NPC*'s.

The process is basically the same in both situations. One player will initiate the trade, then a user interface will appear, which can be similar to the *Inventory* one, but showing the inventory (or partial inventory) of both parties. The two parties can then choose which items to buy, sell, or exchange, and once they both agree on the trade, the changes can be updated on the server.

Strict transaction control is recommended for trading so that there is little chance for exploitation, cheating, or item loss. Care should also be taken when creating temporary items, since this may lead to the duplication of items. For similar reasons, this control should be done on the server, to ensure that one client does not try to exploit another. Again, the authoritative copy of any inventory should live on the server, so that clients can re-read the information from it if they do not agree on the trade.

There are many levels of protection that can be obtained in a trading system, with each higher level involving more control overhead.

A trading system needs protection against hardware and software failures in order to ensure *transaction atomicity, making sure that an item is neither lost nor duplicated*. It means that a transaction in progress does not stop half way if a component in the system fails.

The possible approaches to *item trade* are listed below, in order of increased cost:

Simply swap the items, without performing database *writes*. This approach relies on built-in fault tolerance and disaster recovery, and therefore might have a reasonably large window for item loss or duplication.

- Swap the items and immediately write to the database (two database *writes* per trade). This approach reduces the window for item loss and duplication.
- Use a *trading supervisor* (three database *writes* per trade), leveraging BigWorld's fault tolerance capabilities. This approach results in a high overhead per item.
- Make each item an *entity*.
- Implement external banking-grade database transaction.

Like all forms of insurance, the game implementer must weigh up the importance of losing or duplicating items against the cost of such protection.

3.1. Trade Protocol

Before a trade transaction can take place, both parties must agree on the items being traded. If one of the parties is a NPC character, price tables can be used and this is not an issue. If both parties are players, care must be taken so that players cannot cheat one another.

Player misinterpretation can occur if players are not given the chance to inspect the items offered by their trade partner, or if the player can initiate the transaction after changing his offer once the other has already accepted the trade.

To avoid player misinterpretation problems, the sample implementation uses a trade protocol that allows players to offer, switch, inspect, accept, and refuse items at will until a consensus is reached and the trade can be processed. The protocol is described bellow.

1. The player offers an item. The item is locked locally and the offer is sent to the cell.
   - **In `<res>/scripts/client/Avatar.py`:**

```
def onTradeOfferItem( self, itemSerial ):
    ...
    self._tradeOfferLock = self.inventoryMgr.itemsLock( [itemSerial], 0 )
    self.cell.tradeOfferItemRequest( self._tradeOfferLock, itemSerial )
    ...
```

2. The cell tries to confirm the lock with the base.

   - **In <*res*>/scripts/cell/Avatar.py:**

```
def tradeOfferItemRequest( self, sourceID, lockHandle, itemSerial ):
    ...
    self.tradeOutboundLock = lockHandle
    self.base.itemsLockRequest( lockHandle, [itemSerial], goldPieces )
```

3. The base repeats the lock done on the client and notifies the cell of the result.

   - **In <*res*>/scripts/base/Avatar.py:**

```
def itemsLockRequest( self, lockHandle, itemsSerials, goldPieces ):
    ...
    inventoryMgr = self.inventoryMgr
    ...
    inventoryMgr.itemsRelock( lockHandle, itemsSerials, goldPieces )
    ...
    self.cell.itemsLockNotify( True, lockHandle,
            itemsSerials, itemsTypes, goldPieces )
    ...
```

4. If the locking on the base was successful, the cell notifies the partner about the offer.

   - **In <*res*>/scripts/cell/Avatar.py:**

```
def itemsLockNotify( self, success, lockHandle ... ):
    ...
    partner = self._getModeTarget()
    ...
    partner.tradeOfferItem( itemsTypes[0] )
    ...

def tradeOfferItem( self, itemType ):
    self.tradeSelfAccepted = False
    self.client.tradeOfferItemNotify( itemType )
```

5. The player is notified about the offer made by his trade partner. The item is shown on the user interface.

   - **In /<*res*>scripts/client/Avatar.py:**

```
def tradeOfferItemNotify( self, itemType ):
    ...
    self.inventoryGUI.showOfferedItem( itemType )
    ...
```

6. The player accepts the item offered. The client notifies the cell about the approval.

- **In `<res>/scripts/client/Avatar.py`:**

```
def onTradeAccept( self, accept ):
    ...
    self.cell.tradeAcceptRequest( accept )
```

7. The cell notifies the trade partner about the approval. The notification is immediately forwarded to the client.

- **In `<res>/scripts/cell/Avatar.py`:**

```
def tradeAcceptRequest( self, sourceID, accepted ):
    ...
    self.tradeSelfAccepted = accepted
    partner = self._getModeTarget()
    partner.tradeAcceptNotify( accepted )
    ...

def tradeAcceptNotify( self, accepted ):
    ...
    self.client.tradeAcceptNotify( accepted )
    self.tradePartnerAccepted = accepted
    ...
```

8. The player is notified about the approval by his trade partner. The user interface is updated to reflect that.

- **In `<res>/scripts/client/Avatar.py`:**

```
def tradeAcceptNotify( self, accepted ):
    self.inventoryGUI.tradeOfferAccept( accepted )
```

Note that whenever the player's partner offers a new item, its accepted flag is set to `False` (step 4). This ensures that a trade will never be processed without the explicit approval, from both parties, for the most current item offered.

3.2. Trading Supervisor

Once both parties have agreed on the items to be traded, the trading can take place. The sample implementation makes use of a *trading supervisor* to coordinate and direct the trade.

In summary, the trading supervisor works as follows: each participant in a trade has a *trade id*, which increases monotonically. Before starting a new trade, the supervisor writes and entry for the pending trade into the database. The supervisor then requests the participants to perform the transaction. After completion of the transaction, each participant notifies the supervisor about the trade completed. The entry for pending trade is removed from the database. If the system starts up after a failure, the supervisor lists all pending trades from the database and replays them. The participants use their *trade id*s to determine whether they have carried out the trade. If they have not, the transaction is performed. If they did have, the redundant request is simply ignored.

1. When both avatar have agreed on items, the active cell instructs the base to initiate the transaction

- **In `<res>/scripts/cell/Avatar.py`:**

```
def tradeAcceptRequest( self, sourceID, accepted ):
    ...
```

```
    self._tryTradeBegin()

def _tryTradeBegin( self ):
    if self.mode == Mode.TRADE_ACTIVE and \
        self.tradePartnerAccepted and \
            self.tradeSelfAccepted:
        ...
        self.base.tradeCommitActive( self.tradeOutboundLock )
        partner = self._getModeTarget()
        partner.tradeCommitPassive( self.base )
```

Note

The sample implementation uses the concept active/passive roles in the trade.
The *active* participant will be the one requesting the trade to the trading supervisor.
In the sample, the roles are chosen based on who started the trading mode.
You can use whatever criteria you think is best suited to your design (*e.g.*, higher entity ID).

2. In the base, the avatar in the active participant informs the trading supervisor of the trade ids and the data about items and gold pieces being traded. The data about the trade includes information about each avatar's giveaway: serial number and type ids of items and amount of gold pieces.

   - **In `<res>/scripts/base/Avatar.py`:**

```
...
def tradeSyncRequest( self, partnerBase, partnerTradeParams ):
    ...
    ourItemsTypes, ourGoldPieces = \
        inventoryMgr.itemsLockedRetrieve( self.outItemsLock )

    supervisor = BigWorld.globalBases[ "TradingSupervisor" ]

    selfTradeParams = { "dbID" : self.databaseID, \
                        "tradeID" : self.lastTradeID + 1, \
                        "lockHandle": self.outItemsLock, \
                        "itemsSerials": outItemsSerials, \
                        "itemsTypes": outItemsTypes, \
                        "goldPieces": outGoldPieces }

    if supervisor.commenceTrade( self, selfTradeParams,
            partnerBase, partnerTradeParams ):
    ...
```

3. The *trading supervisor* adds the trading data to its list of pending trades and writes itself to the database.

   - **In `<res>/scripts/base/TradingSupervisor.py`:**

```
...
def commenceTrade( self, A, paramsA, B, paramsB ):
    ...
    tradeLog = { "typeA": A.__class__.__name__, "paramsA": paramsA,
                 "typeB": B.__class__.__name__, "paramsB": paramsB }

    self.recentTrades.append( tradeLog )
    self.outstandingTrades.append( [A.id, B.id] )

    def doTradeStep2( *args ):
```

```
        self._tradeStep2( A, paramsA, B, paramsB )

    self.writeToDB( doTradeStep2 )
    ...
```

4. Once the *trading supervisor* is notified by the database that *write* is complete, it instructs both avatars to modify their inventories, in order to reflect the result of the trade.

   - **In `<res>/scripts/base/TradingSupervisor.py`:**

```
...
def _tradeStep2( self, A, paramsA, B, paramsB ):

    A.tradeCommit(
        self, paramsA[ "tradeID" ], paramsA[ "lockHandle" ],
        paramsA[ "itemsSerials" ], paramsA[ "goldPieces" ],
        paramsB[ "itemsTypes" ], paramsB[ "goldPieces" ] )
    B.tradeCommit(
        self, paramsB[ "tradeID" ], paramsB[ "lockHandle" ],
        paramsB[ "itemsSerials" ], paramsB[ "goldPieces" ],
        paramsA[ "itemsTypes" ], paramsA[ "goldPieces" ] )
```

5. The avatars modify their inventories, increment their *trade id*'s, and write themselves to the database.

   - **In `<res>/scripts/base/Avatar.py`:**

```
...
def tradeCommit(
        self, supervisor, tradeID,
        outItemsLock, outItemsSerials, inItemsTypes, inGoldPieces ):
    TradeHelper.tradeCommit(
        self, supervisor, tradeID, outItemsLock,
        outItemsSerials, outGoldPieces,
        inItemsTypes, inGoldPieces )
```

   - **In `<res>/scripts/base/TradeHelper.py`:**

```
...
def tradeCommit(
        self, supervisor, tradeID,
        outItemsLock, outItemsSerials, inItemsTypes, inGoldPieces ):
    ...
    inItemsSerials = base.inventoryMgr.itemsTrade(
            outItemsSerials, outGoldPieces, inItemsTypes,
            [], inGoldPieces, outItemsLock )
    ...
```

6. Once each avatar is notified by the database that its *write* is complete, it informs the *trading supervisor*.

   - **In `<res>/scripts/base/TradeHelper.py`:**

```
...
def tradeCommit( ... ):
    ...
    base.writeToDB( lambda *args: completeTrade( inItemsSerials ) )
```

7. Once the *trading supervisor* is informed by both parties that their *writes* are complete, it removes the trade from its list of pending trades.

   - **In `<res>/scripts/base/TradingSupervisor.py`:**

```
...
def completeTrade( self, who, tradeID ):
    nost = []
    for t in self.outstandingTrades:
        if t[0] == who.id:
            if t[1] == 0:
                self.recentTrades.pop(len(nost))
                print "TradingSupervisor: trade complete by A"
            else:
                nost.append( (0,t[1]) )
        elif t[1] == who.id:
            if t[0] == 0:
                self.recentTrades.pop(len(nost))
                print "TradingSupervisor: trade complete by B"
            else:
                nost.append( (t[0],0) )
        else:
            nost.append( t )
    self.outstandingTrades = nost
```

Note

After the `write` of a pending trade has been committed into the database (step 3), the trade is guaranteed to be completely carried out on both participant Avatars (step 4 to step 7), even if the system fails at any time during the execution of the transaction.

3.3. Dealing with failure conditions

The system has been designed to cope with component failure. The sample implementation follows the steps below:

1. Upon start up, the *trading supervisor* consult its list of pending trades:

   - **In `<res>/scripts/base/TradingSupervisor.py`:**

```
def __init__( self ):
    ...
    if self.recentTrades == []:
        ...
    else:
        ...
        for trade in self.recentTrades:
            self._replayTrade( trade )
    ...
```

2. The *supervisor* extracts from the database the avatars involved in the trades (in case they have not already been extracted):

   - **In `<res>/scripts/base/TradingSupervisor.py`:**

```
def __init__( self ):
```

```
    ...
    for trade in self.recentTrades:
        self._replayTrade( trade )

def _replayTrade( self, trade ):
    traderMBs = [None, None]
    BigWorld.createBaseFromDBID( trade[ "typeA" ],
        trade[ "paramsA" ][ "dbID" ],
        lambda mb, dbID, wasActive: self.__collectMailbox ( trade[ "typeA"
 ],
            trade, traderMBs, 0, mb ) )
    BigWorld.createBaseFromDBID( trade[ "typeB" ],
        trade[ "paramsB" ][ "dbID" ],
        lambda mb, dbID, wasActive: self.__collectMailbox ( trade[ "typeB"
 ],
            trade, traderMBs, 1, mb ) )

def __collectMailbox ( self, entityType, trade, traderMBs, ind, box ):
    ...
    traderMBs[ ind ] = box
    if traderMBs[ ind^1 ] == None:
        return # still missing other mailbox

    self._tradeStep2( traderMBs[0], trade[ "paramsA" ],
                      traderMBs[1], trade[ "paramsB" ] )
```

3. It then replays the trades, by instructing the avatars to perform it (see step 4 to step 7 in “Trading Supervisor” on page 18).

Note

Overall, the procedure is similar to a journal file system, *i.e.*, the intentions are first written to the database, then the operation is performed, and finally the intentions are marked as complete.

3.4. Creating and destroying the trading supervisor

In the case of a global trading supervisor, the supervisor must be created when the system is first started. In the sample implementation, that is done in the base personality script (`FantasyDemo.py`).

Whenever a new base is started, a three-step procedure is performed to bring up the global `TradingSupervisor`.

1. Check if a global supervisor has not already been created.

   - **In `<res>/scripts/base/FantasyDemo.py`:**

```
import BigWorld
...
def onBaseAppReady( isBootstrap ):
    ...
    TradingSupervisor.wakeupTradingSupervisor()
    ...
```

   - **In `<res>/scripts/base/TradingSupervisor.py`:**

```
def wakeupTradingSupervisor():
```

```
    ...
    if BigWorld.globalBases.has_key( 'TradingSupervisor' ):
        ...
    self:
        doStep2()
    ...
```

2. If not, try to load one from the database (if one is found there, the system has failed, and there can be some trades pending. For more details on replaying trades, see "Dealing with failure conditions" on page 21).

   - **In `<res>/scripts/base/TradingSupervisor.py`:**

```
def wakeupTradingSupervisor():
    ....
    def doStep2():
        BigWorld.createBaseFromDB( 'TradingSupervisor',
                'TradingSupervisor', doStep3 )
    ...
```

3. If none was found in the database, create a new supervisor (the system is coming up after a clean shutdown).

   - **In `<res>/scripts/base/TradingSupervisor.py`:**

```
def wakeupTradingSupervisor():
    ....
    def doStep3( result ):
        if result:
            ...
        else:
            BigWorld.createEntity( 'TradingSupervisor' )
    ...
```

   The fact that this three-step sequence is performed on every base that comes up allow two or more bases that are starting up simultaneously to create one supervisor each (if all get `False` from their query for the global base in step 1). In case two or more supervisors are created, only one of them will be able to register itself globally. The ones that do not will destroy themselves immediately.

   - **In `<res>/scripts/base/TradingSupervisor.py`:**

```
class TradingSupervisor:
    ....
    def __init__( self ):
        ...
        def registerGloballyResult( success ):
            if not success:
                self.destroy()

        self.registerGlobally( self.globalName, registerGloballyResult )
```

   In step 2 in the supervisor startup procedure, the system is assumed to have come from a clear shutdown if the trading supervisor cannot be retrieved from the database. One way to make sure this is always the case is deleting the trading supervisor from the database whenever the system is shutdown.

   - **In `<res>/scripts/base/FantasyDemo.py`:**

```
def destroyTradingSupervisor():
    ....
    dbid = supervisor.databaseID
    supervisor.destroy()
    ...
```

- **In `<res>/scripts/base/TradingSupervisor.py`:**

```
def destroyTradingSupervisor():
    ....
    dbid = supervisor.databaseID
    supervisor.destroy()
```

3.5. Scalability

The example uses only one system-wide *trading supervisor*. However, it is easily scalable to as many as necessary.

Note

Care should be taken as to not create multiple *trading supervisors* on a single `BaseApp` when restoring them in a controlled startup.

A simple way to do this would be to create one trading supervisor instance per BaseApp, and access it through a Python global variable on that `BaseApp`. This could be done in the `onBaseAppReady` callback of the personality script or via an accessor that creates it on first access.

There are also alternatives to how this idea is implemented. For example, the pending trade could be stored with one of the traders. This reduces the number of writes to the database and avoids scalability issues. An extra field would be required per Avatar.

3.6. Vulnerabilities

Despite the high level of protection against crashes afforded by this example solution, there are still conditions in which the trade can go awry.

- **Database corruption**

  If the database corrupts itself, then the list of pending transactions might not be available. Most databases have good protection against this.

- **Scripting errors**

  Care must also be taken to ensure that the inventory is not modified in other unexpected ways. This may include ensuring any items in a trade are not lost before it is complete. For details, see “Locking inventory items” on page 14 .