How To Build a Friends List

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

Most MMOGs allow a player to designate a list of other players as *friends*. These friends are stored in a *friends list*. Players like to be notified when their friends log on or off. The player is usually able to interact with a friend (*e.g.,* chat) directly using the *friends list* without having to be in close proximity to the friend in the game world.

Chapter 2. Features

In this example, we will implement the following features:

- Add and delete a friend by username from a player's *friends list*. The friend does not need to acknowledge that he is being added.
- Player will be notified when his friends log on and off.
- Player can see a list of his friends and their current status (*i.e.*, online or offline).
- Send a message to an online friend regardless of where he is in the game world.
- Request an online friend's health and location, regardless of where he is in the game world.

For details on how to implement other variants of *friends list*, see *Variations* on page 18 .

Note

We are placing a fixed limit on the number of friends a player can have, which influences the design.
The limit is defined by the constant `MAX_FRIENDS` in the base part of Avatar.

Chapter 3. Example Code

The example code is part of the `FantasyDemo` example game.

- In `FantasyDemo`, the player entity is called *Avatar*.
- The MySQL database is required for the *friends list* example to work. It does not work with the XML database.
- The example does not implement any graphical UI elements. It uses the `FantasyDemo` console for user interactions.

3.1. Where should the friends list be stored?

In this example, the *friends list* is part of the player entity (*Avatar*).

Since *friends list* is persistent, it needs to be part of the base of the player entity. But because the list is accessed often in the client, there will be a copy of it in the client part of the player entity as well.

3.2. Admirers

To speed up the notification process, each *Avatar* base also stores a list of *admirers*, *i.e.*, other players who have added the player to their friends list.

Thus, when player logs on or off, *Avatar*'s base can notify all interested parties without having to search through *friends list* of all players in the database.

From player A's perspective:

- The *friends list* is a list of players A is interested in.
- The *admirers list* is a list of players that are interested in A.

Note

UNLIMITED ADMIRERS?
Though there is a limit on the number of *friends* that a player can have, there is no limit on the number of *admirers*. Since each *admirer* must correspond to someone else's *friend*, the maximum total storage cost is:

number of players × max. friends per player.

Since friends are limited for each player, from a system-wide perspective, admirers are limited as well.
Still, there is the possibility that one player is spectacularly popular and has an incredibly large admirers list. This will result in a long delay when the player logs on or off, as Avatar's base notifies player's online status to each of his admirers.
In extreme circumstances, the notification process may block the game on server, and thus have to be processed on a separate thread. For simplicity, we do not handle this case in our example.

3.3. `DBID`

A `DBID` can uniquely identify an *entity* within a *type* (*e.g.*, *Avatar*), whether the *entity* is running or is just an entry in the database.

Since lookup by `DBID` is quicker than by name, *Avatar*'s base also stores *friend*'s `DBID` (alongside its name) in *friends list*.

In fact, the *admirers list* is a list of `DBID`s, since we are not concerned about players' names.

DBIDs are 64-bit integers. They are defined in <res>/scripts/entity_defs/alias.xml for convenience:

```
<root>
   ...
   <DBID>    INT64        </DBID>
   ...
</root>
```

Excerpt of <res>/scripts/entity_defs/alias.xml

3.4. Persistent friends list

We declare the persistent properties of *Avatar* in <res>/scripts/entity_defs/Avatar.def:

```
<root>
   ...
   <Properties>
      ...
      <friendsList>
         <!-- List of tuples: ( string, DBID ) -->
         <Type>       PYTHON    </Type>
         <Flags>      BASE      </Flags>
         <Default>    []        </Default>
         <Persistent> true      </Persistent>
      </friendsList>
      <admirersList>
         <Type>       ARRAY <of> DBID </of> </Type>
         <Flags>      BASE                  </Flags>
         <Default>    []                    </Default>
         <Persistent> true                  </Persistent>
      </admirersList>
      ...
   </Properties>
   ...
</root>
```

Excerpt of <res>/scripts/entity_defs/Avatar.def

The friendsList variable is a Python list of tuples of player's *friends*' names and their DBIDs.

Because the properties are declared as <Persistent>, when the base of the *Avatar* entity is destroyed, the content of the friendsList and admirersList is written to the database. When the base of the *Avatar* entity is re-created, both lists are initialised with data from the database. The lists are initialised with their <Default> value if there is no data in the database.

Note

Why isn't the BASE_CLIENT flag used for friendsList?
It was mentioned earlier in this document that a copy of friends list should be kept on the client part of the Avatar entity because it is frequently accessed there.
So it would seem logical to use the BASE_CLIENT flag, as this would make the list automatically replicated to the client as well.
Unfortunately, this would be true only during initialisation. Changes to friends list during the lifetime of the Avatar object would not be propagated between the client and the base.
Hence, it is made explicit that friendsList variable in the base and the client are separate ones, and that they need to be kept in sync manually.

3.5. Client friendsList

Since the BASE_CLIENT flag is not used, the structure of the friendsList variable on the client does not need to be identical to the one on the base.

Therefore, the client's friendsList variable is implemented as a Python list of tuples (<friend's name>, <whether they are online>), dropping DBID because it is not used in the client.

3.6. Using list indexes as friend identifiers

Even though the friendsList variable in the client and the base are different lists, both have the same order of items in them.

This allows the *Avatar* entity to use the list index as the identifier for a *friend* when communicating between the client and base.

The index will save on communication costs (it is only one byte if a player cannot have more than 256 *friends*) and will allow faster list lookup.

The *friends list* index is defined in <*res*>/scripts/entity_defs/alias.xml for convenience, as well as for consistency, in case you need to implement lists with more than 256 *friends*.

```
<root>
    ...
    <FRIENDIDX>    UINT8    </FRIENDIDX>
    ...
</root>
```

Excerpt of <*res*>/scripts/entity_defs/alias.xml

Note

Indexes and race conditions
Because the Avatar entity is using indexes in communications between the client and the base, both lists have to be kept in sync at all times, otherwise the index in the client may correspond to a different friend than at the base.
This is impossible to achieve in the presence of race conditions.
For example, if someone modifies the list in the base, the base sends an update to the client. But the client sends a message to the base prior to reception of the update message, so it may contain an index that refers to a different friend.
In this document's example, the only operation that can make indexes inconsistent is deleting a friend in the middle of the list. Hence, the Avatar entity could handle the race condition by setting the list entry to "empty", instead of removing it, and then re-using that slot later on. For simplicity, this is not shown.

3.7. Base online friends list

The friendsList variable as defined in Avatar.def does not include any field for storing the online status of a *friend*, *i.e.*, whether he is currently online or offline. This is because the online state is volatile and should not be persistent.

Therefore, the *Avatar* base creates an additional list called friendBases, which is a list of mailboxes to the bases of *friends* that are currently online. The index of items in friendBases and friendsList should match, *i.e.*, friendBases[i] should be the base for the *friend* in friendsList[i]. If the *friend* is not currently online, friendBases[i] is set to None.

This is how you would declare/initialise `friendBases`—as any ordinary Python member variable in `<res>/scripts/base/Avatar.py`:

```
class Avatar( BigWorld.Proxy, AvatarCommon ):

    def __init__( self ):

        self.friendBases = []
```

Excerpt of `<res>/scripts/base/Avatar.py`

Note

`friendBases` not in `Avatar.def`
Note that it is not necessary to declare non-persistent properties in `.def` file.

3.8. Console commands

This example uses the `FantasyDemo` console for user interaction.

We will add the following commands to the console: `addfriend`, `delfriend`, `listfriends`, `msgfriend`, and `infofriend`.

Commands begin with the '`/`' character, so the user would type "`/addfriend simon`" to add *simon* as a *friend*, for example.

For the purposes of this document, typing the command "`/addfriend`" would automatically call the `Avatar.addFriend` function on the client. And `FantasyDemo.addChatMsg( -1, <message>)` outputs a message to the console.

For those interested in more details, look at `<res>/scripts/client/Helpers/ConsoleCommands.py` file and search for its usage in Python code.

3.9. Initialisation and destruction

Described below is the mechanism by which a *friend list* is created and destroyed.

- **The steps during initialisation:**
  - On the base, send the *friends list* to the client.
  - On the base, find out which *friends* are currently online and store their base mailboxes in `friendBases`. Tell the client which *friends* are online.
  - On the base, notify all online *admirers* that the player is currently online.
- **The steps during destruction:**
  - On the base, notify all online *admirers* that the player is going offline.
- **When the player receives a notification from a friend about his online status:**
  - On the base, store the *friend*'s mailbox or None in `friendBases` and forward notification to client.
- **In `<res>/scripts/base/Avatar.py`:**

```
class Avatar( BigWorld.Proxy, AvatarCommon ):
    ...
    def onEntitiesEnabled( self ):
```

```
        ...
        self.initFriendsList()

    def onLoseCell( self ):
        self.notifyAdmirers( False )
        ...

    def initFriendsList( self ):
        # Send list of friend names to client.
        self.client.newFriendsList( [ x[0] for x in self.friendsList ] )

        # Set friend base mailboxes to None (i.e. assume they are offline)
        # Note: Client also assumes friends are offline during
 initialisation.
        self.friendBases =  [ None for x in self.friendsList ]
        for i in range( len(self.friendsList) ):
            BigWorld.lookUpBaseByDBID( "Avatar", self.friendsList[i][1], \
                partial( self.onInitDBLookUpCb, i ) )

        self.notifyAdmirers( True )

    def onInitDBLookUpCb( self, idx, friendBase ):
        if type(friendBase) is not bool:
            # Friend is online
            self.friendBases[idx] = friendBase
            self.client.setFriendStatus( idx, True )

    def notifyAdmirers( self, online ):
        if online:
            ourBase = self
        else:
            ourBase = None
        for admirerDBID in self.admirersList:
            BigWorld.lookUpBaseByDBID( "Avatar", admirerDBID, \
                partial( Avatar_onNotifyAdmirersDBLookUpCb, self.databaseID,
 \
                ourBase ) )

    # friendBase is None if friend is going offline.
    # friendBase is friend's base mailbox if they are coming online
    def onFriendStatusChange( self, friendDBID, friendBase ):
        for i in range( len (self.friendsList ) ):
            if self.friendsList[i][1] == friendDBID:
                self.friendBases[i] = friendBase
                online = friendBase != None
                self.client.setFriendStatus( i, online )
                break

# this callback needs to be a global instead of a method of Avatar because
 we
# notify our admirers when the Avatar is being destroyed.
def Avatar_onNotifyAdmirersDBLookUpCb( ourDBID, ourBase, admirerBase ):
    if type(admirerBase) is not bool:
        # Admirer is online
        admirerBase.onFriendStatusChange( ourDBID, ourBase )
```

<res>/scripts/base/Avatar.py

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

```
class Avatar( BigWorld.Entity ):
    ...
    def __init__( self ):
        ...
        self.friendsList = []

    def newFriendsList( self, friendsList ):
        self.friendsList = [ ( x, False ) for x in friendsList ]

    def setFriendStatus( self, idx, online ):
        friend = self.friendsList[idx]
        self.friendsList[idx] = ( friend[0], online )
        if online:
            FantasyDemo.addChatMsg( -1, friend[0] + " is online." )
        else:
            FantasyDemo.addChatMsg( -1, friend[0] + " has logged off." )
```

<res>/scripts/client/Avatar.py

Note

Maximum message size and method arguments
Remote method calls (*e.g.*, from base to client), are sent as messages.
The maximum size of a message is limited by the UDP packet size. So there is, in fact, a limit on the amount of data (*i.e.*, data contained in the method arguments) that can be passed in a single remote method call.
A call like `self.client.newFriendsList` may be needed to be broken up into multiple calls if the friends list has a large number of items.

3.10. Adding and deleting friends

Described below is the mechanism by which a friend is added or deleted from player's friend list.

- **Steps in adding a friend:**
  - On the client, check that the new *friend* is not already in the list. Tell base to add the *friend*.
  - On the base, check whether the *friend* is currently online. If not online, check whether the *friend* exists in the database. If the *friend* exists, tell him to add player as an *admirer*.
  - On the base, tell the client that the *friend* was added.
  - On the client, add the new *friend* to the list.
- **Steps in deleting a friend:**
  - On the client, find the *friend* in `friendsList` and delete it. Tell the base to delete *friend* by index.
  - On the base, delete the *friend* from `friendsList` and `friendBases`. Tell the *friend* to delete player from his *admirersList*.
- **In `<res>/scripts/base/Avatar.py`:**

```
class Avatar( BigWorld.Entity ):
    ...
    MAX_FRIENDS = 30

    def addFriend( self, friendName ):
```

```
        if friendName == self.playerName:
            self.client.showMessage( 3, 'System',\
                "Adding yourself as a friend is not allowed." )
        if len(self.friendsList) >= self.MAX_FRIENDS:
            self.client.showMessage( 3, 'System',\
                "You already have the maximum number of friends allowed: " \
                + str(self.MAX_FRIENDS) )
        else:
            BigWorld.createBaseFromDB( "Avatar", friendName, \
                partial( self.onAddFriendCreateBaseCb, friendName ) )

    def onAddFriendCreateBaseCb( self, friendName, friendBase, dbID, \
                                 wasActive ):
        if friendBase != None:
            if wasActive:
                friendBase.addAdmirer( self.databaseID, self, False )
            else:
                # addAdmirer() needs playerName to be set.
                friendBase.playerName = friendBase.cellData[ "playerName" ]
                friendBase.addAdmirer( self.databaseID, self, False )
                # Should destroy the base that we've created temporarily
                friendBase.destroy()
        else:
            self.client.showMessage( 3, 'System',
                "Cannot add unknown player: " + friendName )

    def onAddedAdmirerToFriend( self, friendName, friendDBID, friendBase ):
        # Double check here due to race condition of multiple addFriends
        # when we have MAX_FRIENDS - 1 friends
        if len(self.friendsList) >= self.MAX_FRIENDS:
            self.client.showMessage( 3, 'System', \
                "You already have the maximum number of friends allowed: " \
                + str(self.MAX_FRIENDS) )
        else:
            self.friendsList.append( ( friendName, friendDBID ) )
            self.friendBases.append( friendBase )
            online = friendBase != None
            self.client.onAddedFriend( friendName, online )

    def delFriend( self, friendIdx ):
        friendBase = self.friendBases.pop(friendIdx)
        if friendBase != None:
            friendBase.delAdmirer( self.databaseID )
        else:
            BigWorld.createBaseFromDBID( "Avatar", \
                self.friendsList[friendIdx][1], self.onDelFriendCreateBaseCb
)
        del self.friendsList[ friendIdx ]

    def onDelFriendCreateBaseCb( self, friendBase, dbID, wasActive ):
        if friendBase != None:
            friendBase.delAdmirer( self.databaseID )
            # Should destroy the base that we've created temporarily
            friendBase.destroy()

    def addAdmirer( self, admirerDBID, admirerBase, online ):
        self.admirersList.append( admirerDBID )
        if online:
            onlineBase = self
        else:
            onlineBase = None
```

```
        admirerBase.onAddedAdmirerToFriend( self.playerName,
 self.databaseID, \
            onlineBase )

    def delAdmirer( self, admirerDBID ):
        self.admirersList.remove( admirerDBID )
```

<*res*>/scripts/base/Avatar.py

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

Note

We have added additional code on the client to get the name of the player currently targeted, in the case that it is not specified in the console command.

```
# Helper method to get the target player name if friendName is empty
    def getTargetForFriendlyAction( self, friendName ):
        if len(friendName) == 0:
            target = BigWorld.target()
            if target != None and isinstance(target, Avatar):
                return target.playerName
            else:
                FantasyDemo.addChatMsg( -1, \
                    "Please specify friend name or have friend targetted." )
                return ""
        else:
            return friendName

    # Helper method to find the index of friendName in self.friendsList
    def getFriendIdxByName( self, friendName ):
        for i in range( len(self.friendsList) ):
            if self.friendsList[i][0] == friendName:
                return i
        return -1

    def addFriend( self, friendName ):
        targetFriendName = self.getTargetForFriendlyAction(friendName)

        if len(targetFriendName) > 0:
            idx = self.getFriendIdxByName(targetFriendName)
            if idx < 0:
                self.base.addFriend( targetFriendName )
            else:
                FantasyDemo.addChatMsg( -1, targetFriendName + \
                    " is already your friend." )

    def onAddedFriend( self, friendName, online ):
        self.friendsList.append( ( friendName, online ) )
        FantasyDemo.addChatMsg( -1, friendName + " is your new friend." )

    def delFriend( self, friendName ):
        targetFriendName = self.getTargetForFriendlyAction(friendName)

        if len(targetFriendName) > 0:
            idx = self.getFriendIdxByName(targetFriendName)
            if idx >= 0:
                del self.friendsList[idx]
                self.base.delFriend(idx)
                FantasyDemo.addChatMsg( -1, targetFriendName + \
```

```
                    " is no longer your friend." )
            else:
                FantasyDemo.addChatMsg( -1, targetFriendName + \
                    " is not currently one of your friends." )

    # We received a message
    def showMessage( self, type, source, msg ):
        FantasyDemo.addChatMsg( -1,
            ( "Debug", "Tell", "Group", "Info" )[type] + " - " + source + ":
" + msg )
```

<res>/scripts/client/Avatar.py

Note

Local bases: Synchronous method calls and other goodies
When bases are on the same BaseApp, method calls are synchronous. That is why `onAddFriendCreateBaseCb` and `onDelFriendCreateBaseCb` are able to destroy `friendBases` straight after calling one of its methods.
In this case, `friendBases` is guaranteed to be on the same BaseApp because it was created using `BigWorld.createBaseFromDB`, and was not already active.
Plus, local bases can have their properties and methods accessed, even if not declared in the `.def` file.
So, in principle, `onAddFriendCreateBaseCb` would not have to call the `addAdmirer` method and receive a call to the `onAddedAdmirerToFriend` method, because it would be able access `friendBase.databaseID` directly. However, this is would mean code duplication.
The downside to these "goodies" is that it could mask bugs during initial testing, when usually only one BaseApp is running, which forces all bases to be local.

Note

Dealing with unreliable remote base method calls
In general, base method calls are not any more unreliable than other components. But because the BigWorld Server relies on bases for data persistence, things can go wrong when base methods calls are not executed.
This can happen when the remote base is destroyed during message transit (in which case the method call might be silently ignored), or when base data changed by a call is reverted (for example, in the event a BaseApp crash occurs and a backup BaseApp takes over).
The example in this document is mainly concerned with the `addAdmirer` and `delAdmirer` calls. If these are ignored or rolled back, player could end up with inconsistent friends and admirers lists between bases (*e.g.*, John thinks Simon is his friend, but Simon does not have John in his admirers list).
This situation could be dealt with by adding self-correcting code in `Avatar_onNotifyAdmirersDBLookUpCb` to remove the admirer from player's `admirersList` if player is not one of his friends.
Similarly, self-correcting code could be added to `onFriendStatusChange` to fix the friend's `admirersList` if he is not a friend of the player.
Alternatively, the design could have be changed to have a dedicated base or database table responsible for storing the friends and admirers list for all players. Player entities (Avatars) would then talk to this base (or database) to update their `friendsList`.

3.11. Interacting with friends

Interacting with *friends* is not very different to interacting with other players.

In fact, the following methods could have been implemented as general methods for interacting with any player (identified by his username). The advantage of limiting these interactions to *friends* is efficiency. Since the *Avatar* entity already knows their online status and has a copy of their base mailboxes, it can make smarter decisions (*e.g.*, reject the operation when target player is not online) and avoid accessing the database.

- **When sending a message to a friend:**
  - On the client, check that the *friend* is in `friendsList`. Tell base to send the message to the *friend*.
  - On the base, check that the *friend* is online. Tell the *friend*'s base to send the message to his client.
- **When getting info on a friend:**
  - On the client, check that the *friend* is in `friendsList`. Tell the base to get info on the *friend*.
  - On the base, check that the *friend* is online. Tell the *friend*'s base to get info on him.
  - On the *friend*'s base, get info for the *admirer* and tell our cell to get info for the *admirer*.
  - On *friend*'s cell, get info for the *admirer* and tell the *admirer*'s base to send that info to his client.

  An extra access to the cell is made because that is where most of the information is.
- **When getting list of friends:**
  - On the client, display names and online status from `friendsList`.
- **In `<res>/scripts/base/Avatar.py`:**

```
class Avatar( BigWorld.Entity ):
    ...
    def sendMessageToFriend( self, friendIdx, message ):
        friendBase = self.friendBases[friendIdx]
        if friendBase != None:
            friendBase.client.onReceiveMessageFromAdmirer( self.playerName,
 \
                message )
        else:
            self.client.showMessage( 3, 'System',
                "Cannot send message to offline player." )

    def getFriendInfo( self, friendIdx ):
        friendBase = self.friendBases[friendIdx]
        if friendBase != None:
            friendBase.getInfoForAdmirer( self )
        else:
            self.client.showMessage( 3, 'System', \
                "Cannot get information on offline player." )

    def getInfoForAdmirer( self, admirerBase ):
        friendNames = [ name for (name, dbid) in self.friendsList ]
        self.cell.getInfoForAdmirer( \
            "[Friends: " + str(friendNames)[1:-1] + "]", admirerBase )
```

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

```
class Avatar( BigWorld.Entity ):
    ...
    def getInfoForAdmirer( self, baseInfo, admirerBase ):
```

```
        info =  "[Health: " + str(self.healthPercent) + "%]"
        info += "[Frags: " + str(self.frags) + "]"
        info += "[Position: " + str(self.position) + "]"
        admirerBase.client.onRcvFriendInfo( self.playerName, info + baseInfo
)
```

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

```
    def infoFriend( self, friendName ):
        targetFriendName = self.getTargetForFriendlyAction(friendName)

        if len(targetFriendName) > 0:
            idx = self.getFriendIdxByName(targetFriendName)
            if idx >= 0:
                self.base.getFriendInfo(idx)
            else:
                FantasyDemo.addChatMsg( -1, targetFriendName + \
                    " is not one of your friends." )

    def onRcvFriendInfo( self, friendName, info ):
        FantasyDemo.addChatMsg( -1, info )

    def listFriends( self ):
        FantasyDemo.addChatMsg( -1, "You have " + str(len(self.friendsList))
+ \
            " friend(s):" )
        onlineFriends = [ name for (name, online) in self.friendsList \
                          if online ]
        onlineFriendsStr = "   online:" + str(onlineFriends)[1:-1]
        FantasyDemo.addChatMsg( -1, "   online: " + str(onlineFriends)[1:-1]
)
        offlineFriends = [ name for (name, online) in self.friendsList \
                           if not online ]
        FantasyDemo.addChatMsg( -1, "   offline: " +
str(offlineFriends)[1:-1] )

    def msgFriend( self, friendName, message ):
        targetFriendName = self.getTargetForFriendlyAction(friendName)

        if len(targetFriendName) > 0:
            idx = self.getFriendIdxByName(targetFriendName)
            if idx >= 0:
                self.base.sendMessageToFriend( idx, message )
                FantasyDemo.addChatMsg( -1, "You say to " + targetFriendName
+ \
                    ": " + message )
            else:
                FantasyDemo.addChatMsg( -1, targetFriendName + \
                    " is not one of your friends." )

    def onReceiveMessageFromAdmirer( self, admirerName, message ):
        FantasyDemo.addChatMsg( -1, admirerName + ": " + message )
```

Note

Special remote entity properties
Though it is not generally possible to access a remote entity's properties, it is possible to access its mailboxes to other parts of itself (*i.e.*, `base.client`, `base.cell`, `cell.client`, `cell.base`).
Hence, the call `friendBase.client.onReceiveMessageFromAdmirer` in base method `sendMessageToFriend` and the call `admirerBase.client.onRcvFriendInfo` in cell method `getInfoForAdmirer` are perfectly legitimate.

3.12. Declaring the methods

Methods must be declared in the `.def` file if they are called remotely (*i.e.*, calls between client and base), between bases, between cell and base, etc. Furthermore, methods called by the client must be declared `<Exposed/>`.

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

```
<root>
    ...
    <ClientMethods>
        ...
        <newFriendsList>
            <Arg>        ARRAY <of> STRING </of> </Arg> <!-- array of friend
 names -->
        </newFriendsList>
        <onAddedFriend>
            <Arg>        STRING    </Arg>    <!-- friend's name -->
            <Arg>        BOOL      </Arg>    <!-- is friend online? -->
        </onAddedFriend>
        <setFriendStatus>
            <Arg>        FRIENDIDX </Arg>    <!-- friend's list index -->
            <Arg>        BOOL      </Arg>    <!-- is friend online? -->
        </setFriendStatus>
        <onReceiveMessageFromAdmirer>
            <Arg>        STRING    </Arg>    <!-- admirer's name -->
            <Arg>        STRING    </Arg>    <!-- message -->
        </onReceiveMessageFromAdmirer>
        <onRcvFriendInfo>
            <Arg>        STRING    </Arg>    <!-- friend's name -->
            <Arg>        STRING    </Arg>    <!-- friend's info -->
        </onRcvFriendInfo>
        <showMessage>
            <Arg>        UINT8     </Arg>    <!-- type of message -->
            <Arg>        STRING    </Arg>    <!-- source of message -->
            <Arg>        STRING    </Arg>    <!-- message to show on console
 -->
        </showMessage>
    </ClientMethods>

    <BaseMethods>
        ...
        <addFriend>
            <Exposed/>
            <Arg>        STRING    </Arg>    <!-- friend's name -->
        </addFriend>
        <delFriend>
            <Exposed/>
```

```
            <Arg>        FRIENDIDX </Arg>    <!-- friend's list index -->
        </delFriend>
        <addAdmirer>
            <Arg>        DBID      </Arg>    <!-- admirer's dbid -->
            <Arg>        MAILBOX   </Arg>    <!-- admirer's base -->
            <Arg>        BOOL      </Arg>    <!-- are we online? -->
        </addAdmirer>
        <onAddedAdmirerToFriend>
            <Arg>        STRING    </Arg>    <!-- friend's name -->
            <Arg>        DBID      </Arg>    <!-- friend's dbid -->
            <Arg>        MAILBOX   </Arg>    <!-- friend's base -->
        </onAddedAdmirerToFriend>
        <delAdmirer>
            <Arg>        DBID      </Arg>    <!-- admirer's dbid -->
        </delAdmirer>
        <onFriendStatusChange>
            <Arg>        MAILBOX   </Arg>    <!-- friend's base -->
            <Arg>        BOOL      </Arg>    <!-- is friend online? -->
        </onFriendStatusChange>
        <sendMessageToFriend>
            <Exposed/>
            <Arg>        FRIENDIDX </Arg>    <!-- friend's list index -->
            <Arg>        STRING    </Arg>    <!-- message -->
        </sendMessageToFriend>
        <getFriendInfo>
            <Exposed/>
            <Arg>        FRIENDIDX </Arg>    <!-- friend's list index -->
        </getFriendInfo>
        <getInfoForAdmirer>
            <Arg>        MAILBOX   </Arg>    <!-- admirer's base -->
        </getInfoForAdmirer>
    </BaseMethods>

    <CellMethods>
        ...
        <getInfoForAdmirer>
            <Arg>        STRING    </Arg>    <!-- our base info for admirer
-->
            <Arg>        MAILBOX   </Arg>    <!-- admirer's base -->
        </getInfoForAdmirer>
    </CellMethods>
```

<res>/scripts/entity_defs/Avatar.def

Chapter 4. Variations

The example here is a starting point for building your own *friends list*. We have pointed out areas that need further work before it would be suitable for use in a production system. Certainly you will also have specific requirements for your game. This section discusses some obvious variations and their implications.

4.1. Asking for permission to be someone's friend

This could be a requirement if *friends* have special access (*e.g.*, the `getInfo` command in this document's example).

Ideally, the *friend* can subsequently rescind his friendship after granting it. This means *admirers list* should become editable by the user. Hence, it should probably store more than just the `DBID` for performance reasons.

It may also be a good idea then to limit the size of the *admirers list*, since there would be usability problems when this list is large.

Each *"Can I be your friend?"* request should also be stored somewhere. Places they could be stored:

- **On the client of the player being asked (the askee).**
  - This is the bare minimum, since the *askee* must have visual indication of being asked.
  - When the *askee* logs off, requests are forgotten.
  - The *asker* cannot keep track of outstanding requests.
  - A possible situation exists where acceptance is rejected because the *asker*'s *friends list* has become full since the request was made.
- **On the client of the askee and on the base of the player who is asking (the asker).**
  - The *asker* can keep track of outstanding requests, therefore will not exceed *friends list* limit.
  - The *asker* can withdraw the request.
  - The *askee* can notify the *asker* when he logs off (by using info in the request from the *asker*) and the *asker* can remove outstanding requests.
  - It is possible, but it does not make much sense for the *asker* to persist requests because *askee* will forget requests when he logs off.
- **On the client and base of the askee, and on the base of the asker.**
  - Same as above except...
  - Both *askee* and *asker* can persist requests, and requests can be accepted independent of each other's online status.
  - Possible data integrity problems because we have persistent data in two entities that needs to be kept in sync.

4.2. Enforcing mutual friendship

Where mutual friendship is enforced, it will be possible to eliminate the separate *friends list* and *admirers list*.

So when player A adds player B into his *friends list*, he is automatically added to B's *friends list*.

It is highly recommended that players be forced to ask permission before forming friendships (see above). This is to prevent total strangers from using up the limited number of friends each player is allowed to have.

Note that enforcing mutual friendship would not allow a game world to have "leader" characters with many more admirers than friends.

4.3. Matching request and response

In this document's example, the base would often call `self.client.showMessage(<type>,<source>,<message>)` as response to a request from the client.

Often this is not sufficient for real GUIs. For example, a button may be disabled while the operation is in progress and re-enabled when the operation completes. So the client must know that the `showMessage` corresponds to an outstanding request, and not some unsolicited message (*e.g.*, message from a friend).

One solution would be to make a separate callback method for each operation (*e.g.*, `onAddedFriend`, `onDeletedFriend`, etc). But this could make the `.def` file quite cluttered.

Another solution would be to pass *sequence numbers* and have them passed back in the result (*e.g.*, `showMessage(<sequence number>,<type>,<source>,<message>)`). This would have the advantage of supporting multiple outstanding requests of the same type (*e.g.*, multiple `addFriends`), but is obviously more expensive (extra integer in message and dictionary in the client to map *sequence number* to request) and therefore should be limited to infrequent calls.