SFS2X Space Race

Top-Down Multi-Player Race

DOCUMENTATION / WALK-THROUGH

Release Date: April 2011
Written By: Wayne Helman

SFS2X Space Race Walk-Through

SmartFoxServer is a robust and mature platform for developing multi-user applications. Written in Java, the server supports clients written on a number of different platforms including Android, iOS, Unity, and Flash.

SmartFoxServer 2X introduces new features and enhanced performance compared to SmartFoxServer 1.x. One such feature is the introduction of the UDP protocol accessible through server extensions. The following walk-through tutorial and sample application describes a full working top-down race game that utilizes some commonly used features of SmartFoxServer and makes extensive use of the UDP protocol.

Our game will be built in Adobe AIR 2 using ActionScript 3. Currently, the UDP protocol is only available in AIR 2. Hopefully, future versions of Flash will also support it.

UDP (User Datagram Protocol) vs TCP (Transmission Control Protocol)

TCP is the most common protocol deployed on the Internet. It's dominance is explained by the fact that TCP performs error correction. When TCP is used, there is an assurance of delivery. This is accomplished through a number of characteristics including ordered data transfer, retransmission, flow control, and congestion control. During the delivery process, packet data may collide and be lost, however TCP ensures that all packet data is received by re-sending requests until the complete package is successfully delivered.

UDP as well, is commonly found on the Internet, however, UDP is not used to deliver critical information – it forgoes the data checking and flow control found in TCP. For this reason, UDP is significantly faster and more efficient, although, it cannot be relied on to reach it's destination.

For SmartFoxServer, UDP offers the ability to update other players with non-critical information quickly and frequently. In our sample application, we will be using it for player positioning and verification.

Setting Up The SmartFoxServer Zone

We're going to assume you have a running SmartFoxServer 2X already installed either locally or on a remote server. Perform the following steps to install the included zone:

1. Copy the *spaceRace.zone.xml* file to your server's */SFS2X/zones* directory.
2. Copy the */SpaceRace* folder with the extension JAR files to your server's */SFS2X/extensions* directory.
3. Restart the server.
4. Launch the SFS2X Administration Tool to verify that the zone has been successfully loaded.

Setting Up FlashBuilder for AIR 2.0

To get started, FlashBuilder requires some configuration to enable compiling to AIR 2. First, download the latest Flex SDK from Adobe http://opensource.adobe.com/wiki/display/flexsdk/Download+Flex+4.5 (currently 4.5) and AIR SDK http://www.adobe.com/products/air/sdk/ (currently 2.6).

The AIR SDK must be overlaid on top of the Flex SDK. Instructions on how to accomplish this can be found here: http://kb2.adobe.com/cps/495/cpsid_49532.html.

Please note: future version of the Flex SDK will bundle the AIR SDK. A version of this SDK is already available

at http://opensource.adobe.com/wiki/display/flexsdk/Download+Flex+Hero.

More information is available in the *Readme.txt* file included in the ZIP archive.

Once you have configured the SDK, import the project into FlashBuilder and configure the project to use the new SDK.

1. *File -> New -> Flex Project*
2. Give your project a name eg. SpaceRace
3. Select where your project is going to be located
4. Make sure you have Application Type selected as Desktop (runs in adobe AIR)
5. Hit next twice
6. Hit finish
7. Flex builder generates your project folders (bin-debug, libs, src)
8. Paste the client src folder from the zip file in your project src folder
9. Double check if you have all the swc files necessary inside libs

You should now have the project installed and visible in the Package Explorer. Right click on the project and select *Properties*. Navigate to *Flex Compiler*, and select the newly installed SDK we created earlier.

Open up */src/SFS2X_SpaceRace-app.xml*. Line 2 should read:

```
<application xmlns="http://ns.adobe.com/air/application/2.0">
```

This identifies the target application version. When creating a new AIR application, Flash Builder occasionally defaults to version 1.5. When creating a new project for SmartFoxServer and you are planning to use UDP, be sure to change the version number to 2.0.

Finally, run debug to verify that the application is working and all configurations are set properly.

Setting Up Eclipse For Java Extensions

1. *File -> New -> Java Project*
2. Type a Name for the project like "*SFS2XSpaceRaceZoneExt*"
3. Select *Create project from existing source*, and navigate to the extracted Java source code included in the download.
4. Click on *Finish* to import the project and follow the prompts.

Tutorial Conventions

Keeping code style consistent is an important factor when working with a team and even more important when teaching and sharing. In our sample code we use some conventions that should be defined prior to delving into the ActionScript code.

Class properties always begin with an underscore e.g.

```
private var _someVar:String = 'foobar';
```

Method arguments always begin with the dollar symbol e.g.

```
private function handleLogin($event:MouseEvent):void
```

Local variable have no prefix e.g.

```
var username:String = 'mr.foo';
```

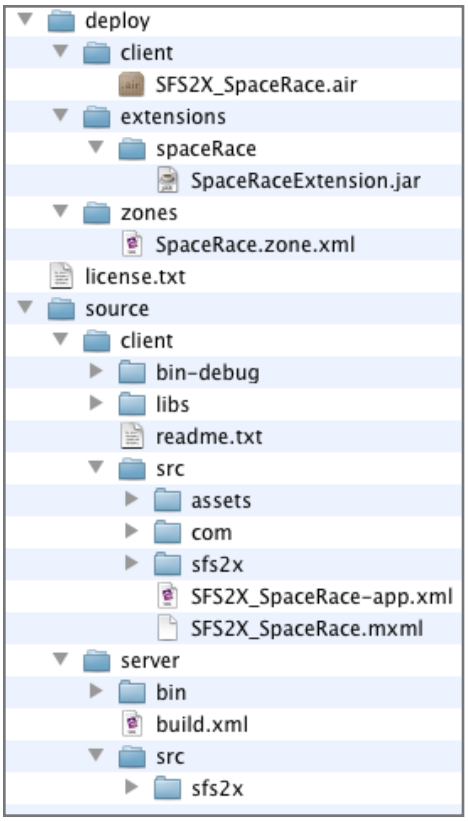
Multiple declarations are separated by a comma e.g.

```
var username        :String      = 'mr.foo',
    password        :String      = 'razzamataz',
    userId          :int         = 244;
```

Limitations Developing Multi-User Apps with Adobe AIR

Developing multi-user application for AIR poses one major problem that Flash applications do not experience. For development and testing, Flash is capable of running multiple instances of an application on a single system. With AIR, only a single instance of an application can be run on a single machine. You cannot even run one installed application and try to run the same application in the FlashBuilder IDE. This limitation makes it very difficult to develop multi-user applications since testing your application requires two physically separate computers or a host running a virtualized second instance of an operating system.

Because of this limitation, we've developed a system within a single AIR application that makes use of the `NativeWindow()` class to instantiate multiple instances of our game in a single AIR application. There are however issues that arise using this method.



Code Structure

To facilitate multiple instances of our game in a single AIR application, we need to label each window we instantiate and pass that label down through our class structure. When a class needs to refer to the *Stage*, or reference a Singleton pattern, we need to first pass in our label to identify which `NativeWindow()` we are referring to. Later in this tutorial we'll delve deeper into how that is accomplished.

The one unfortunate side effect of this structure is that we need to avoid static classes in our game code for anything that needs to be unique to a game instance. For the sake of simplicity, we are using an animation engine that is static, and in that case, we have to avoid calling methods like "*killAllTweens*", since that will affect all windows and possibly destroy animations in another game instance.

There are some static classes in the client side application, most

notably the `Settings()` and `InstanceManager()` classes. The `Settings()` class defines all of our static constants that we'll use in our application and is a single source for us to store our configuration options. The `InstanceManager()` manages two types of singletons enabling us to have multiple games running simultaneously. More on this as we dive into the code.

Since AIR is intended to be installed on a system, we have avoided any kind of loader class. All configurations, even CSS definitions are available to us in our static `Settings()` class. For imagery, we included an `Assets()` class that loads and stores a class reference to each image. When we need an image we simply instantiate a new instance of the image class.

The same concept is applied to configuring our SmartFoxServer connection properties. With SmartFoxServer 2X, the built in `loadConfig()` method typically would read from an XML file and call `connect()` behind the scenes. Since we want to avoid an additional XML file to load and maintain all configurations options in a single file, we'll call `connect()` directly.

Diving In

Let's start by opening up the *SFS2X_SpaceRace.mxml* file in the root of the */src* folder and the *sfs2x.games.spacerace.config.Settings* class.

[AS3 Main line 78]

```
if(Settings.DEBUG)
{
    for(var i:int = 1; i < Settings.DEBUG_WINS + 1; i++)
    {
        spawnAppWindow('debug' + i);
    }
}
```

The constructor in *Main.as* first creates our main application window and adds a game to the stage. Subsequently, it checks our *Settings.as* class to see if `DEBUG` is set to `true`. If so, we can add additional windows that will enable us to test as if there are two or more players. The number of extra windows is defined in `Settings.DEBUG_WINS`. Setting `DEBUG` to true will also display some hidden and extra elements on the stage.

The `spawnAppWindow()` method will create additional application windows by instantiating new `SFSSpaceRace()` classes and assign a `windowsID` (type String) to be referenced whenever we need to make a call specifically to that instance of the game. We're using the default window title to identify each game instance, so the main application window is named "SFS2X Space Race", whereas additional windows are named "debug1", "debug2" and so on.

Each `SFSSPaceRace()` instance will set up our game environment by creating our space background, retrieving an instance of a SmartFoxServer wrapper and finally displaying the log on screen. To manage instances of some classes, we're using the `InstanceManager()` class. This class has one purpose – that is, to return instances of other classes as a singleton would. Again, since we require "multiple" single instances to accommodate multiple game windows, we need to pass in the `windowID` property of the game instance to the `InstanceManager()` so it can assign an instance for that specific window and return it upon request. When using the `InstanceManager()` to retrieve our instance of `SFSWrapper()`, we can register event listeners in numerous classes and perform logic where we need it. The majority of SmartFox listeners exist in `SFSWrapper()` and `SFSSpaceRace()`, and our two controllers, `LobbyController()` and `GameController()`.

The `SFSWrapper()` class handles much of the default SFS2X functionality for establishing a connection and login to the server. It extends *com.smartfoxserver.v2.SmartFox* (the base SFS2X AS3 client class) and simply adds some additional functionality to handle connection events.

Once our environment has been instantiated, we can display a log in screen to the user. Defaults populate the form retrieved from our `Settings()` class. Please be sure to change the `Settings()` constants to refer to your particular server environment.

Once the form data is submitted through the login window, a series of events occur. First, the `handleLogin()` method is called:

[AS3 sfs2x.games.spacerace.ui.windows.WinLogin line 284]

```
private function handleLogin($e:MouseEvent = null):void
{
    var wrapper:SFSWrapper = InstanceManager.getSFSInstance((this.parent as SFSSpaceRace).windowID);
    wrapper.setUserName = _un.text;
    wrapper.setIP = _ip.text;
    wrapper.setTCPPort = parseInt(_tcp.text);
    wrapper.setUDPPort = parseInt(_udp.text);
    wrapper.setZone = _zone.text;
    wrapper.connectToServer();
}
```

You'll notice here, that we retrieve an instance of the `SFSWrapper()` by calling our `InstanceManager()` and passing in the `windowID` property of our parent class, `SFS2XSpaceRace()`. Next we assign properties to the `SFSWrapper()` with the values from the text fields of the form. Again, as mentioned before, we're avoiding using the default SFS2X method loadConfig() in favour of assigning values manually. This avoids the need to load an external XML file.

The values that are set are typical SmartFoxServer connection properties with the addition of the UDP port. Since we'll be using both TCP and UDP messaging, we are required to define a port for each protocol.

Once all properties have been set, we can establish a connection with the server and handle a successful connection or a failed one from within the `SFSWrapper()`. The connection to SFS2X will occur over the TCP port. UDP communication will need to be initiated once we have established a connection.

If we are successful at connecting to the server, we can then call our login method:

[AS3 sfs2x.games.spacerace.SFSWrapper line 125]

```
private function login():void
{
    var request:LoginRequest = new LoginRequest(_username, '', _zone);
    send(request);
}
```

The log in method creates a new *Login Request*. *Requests* of this kind are new to SFS2X and offer greater flexibility and a more semantic approach to calling SFS2X server side methods. The *Login Request*, as with all

other Requests, extends `BaseRequest()` in the SFS2X AS3 API. This enables our `SFSWrapper()` class and by definition the core `SmartFox()` class to send requests to the SFS2X server in a similar manner.

The *Login Request* takes four arguments, `userName`, `password`, `zoneName`, and `params`. For our sample application, we're leaving the password blank and not passing any additional values as parameters. For a more detailed look at *Requests* and specifically the `LoginRequest()`, refer to the SFS2X docs at: http://docs2x.smartfoxserver.com/api-docs/asdoc/com/smartfoxserver/v2/requests/LoginRequest.html

When SFS2X receives this request, it will create a new User Object in the zone and return our User's name and ID.

[AS3 sfs2x.games.spacerace.SFSSpaceRace line 269]

```
private function onLogin($e:SFSEvent):void
{
	_sfs.removeEventListener(SFSEvent.LOGIN, onLogin);
	_sfs.removeEventListener(SFSEvent.LOGIN_ERROR, onLoginError);

	Settings.setGlobal('user_name', $e.params.user.name, _windowID);
	Settings.setGlobal('user_id', $e.params.user.id, _windowID);
	Settings.setGlobal('root', this, _windowID);

	_udpManager = new AirUDPManager();

	_sfs.initUDP(_udpManager, _sfs.getIP, _sfs.getUDPPort);
}
```

Our `Settings()` class contains two methods for storing and retrieving data for frequent use – `setGlobal()` and getGlobal(). Again, we need to store references per window, so both those methods take `windowID` as an argument. Using `setGlobal()`, we'll store our user ID and user name for later reference. Once that is complete, since we are going to be using UDP, we need to instantiate the SFS2X `AirUDPMananger()` and initialize it with our `SFSWrapper()` instance.

The timing in which you initialize the UDP connection is important. It should only be initialized after successful connection and login to the zone. As with other server methods, the AS3 client can listen for an event (`SFSEvent.UDP_INIT`) that is dispatched on successful initialization.

Once we've verified our UDP connection, our game lobby is then constructed.

Game Architecture

SmartFoxServer 2X has all the available features it's predecessor had plus more. For our game, we're going to be using a number of built in SFS2X functionality:

User Variables – for status updates
Room Variables – for spacecraft selection and initial positioning of track objects
Global Room Variables (new in SFS2X) – for game start notification
TCP Extensions – for health, advantages, off track indicator
UDP Extensions (new in SFS2X) – for spacecraft position updates and gun fire

Public Messages – for open chat in the *Lobby*
Private Message – for private chat in pre-game mode
Object Messages – for poking and broadcasting keyboard events

One of the first things you will notice when joining the *Zone*, is that the standard user list is not used, but rather, we request a custom user list from the server. This user list includes the *User* objects of all connected users. In the vast majority of applications, this is not recommended. Typically, we want to reduce the size of all messages being transmitted to and from the server. Making a call to retrieve all users is certain to cause issues in a production environment where concurrent users can reach into the hundreds or thousands. Nevertheless, it is included here for illustrative purposes.

The players list in the lobby provides a status indicator of what each connected user is doing. This is handled with *User Variables*, however, we've written a Java Extension to capture the `USER_VARIABLES_UPDATE` event and broadcast the event only to the *Lobby* regardless of whether it occurs in a game room. This allows us to let the players in the *Lobby* update the status of everyone in the player list. The Java class for this is *sfs2x.extensions.games.spacerace.handlers.OnUserVarChange*.

As players enter the *Zone*, they set their own status *User Variable* to *In Lobby*.

To create a new game, players can enter a name in the provided field, and click either *Create Game* or *Practice*. Once a button is clicked, a game creation request is sent to the server to create a game room.

[AS3 sfs2x.games.spacerace.LobbyController line 531]

```
private function addRoom($e:MouseEvent = null):void
{
    var settings    :SFSGameSettings    = new SFSGameSettings(_roomName.text),
        myRoomVarObj:ISFSObject         = new SFSObject(),
        roomVarObj  :ISFSObject         = new SFSObject(),
        slot1       :SFSRoomVariable,
        //since I'm creating the room, I assume slot 1 (first player), created below
        slot2       :SFSRoomVariable    = new SFSRoomVariable('2', roomVarObj),
        //slot 2 (second player), empty
        slot3       :SFSRoomVariable    = new SFSRoomVariable('3', roomVarObj),
        //slot 3 (third player), empty
        slot4       :SFSRoomVariable    = new SFSRoomVariable('4', roomVarObj),
        //slot 4 (fourth player), empty
        events      :RoomEvents         = new RoomEvents();

    myRoomVarObj.putUtfString(Settings.PLAYER_NAME, _sfs.mySelf.name);

    slot1 = new SFSRoomVariable('1', myRoomVarObj);

    events.allowUserVariablesUpdate = true;
    events.allowUserExit = true;
    events.allowUserEnter = true;
    events.allowUserCountChange = true;

    settings.maxVariables = 7;
    settings.notifyGameStarted = true;
    settings.maxUsers = 4;
    settings.isPublic = true;
    settings.minPlayersToStartGame = 2;
```

```
    settings.isGame = true;
    settings.groupId = 'games';
    settings.variables = [ slot1, slot2, slot3, slot4 ];
    settings.extension = new RoomExtension('SpaceRace',  'sfs2x.extensions.games.spacerace.
SpaceRaceGameExtension');
    settings.leaveLastJoinedRoom = true;
    settings.events = events;

    var request:CreateSFSGameRequest = new CreateSFSGameRequest(settings);
    _sfs.send(request);
}
```

Here, we create our game room settings, set our default *Room Variables* and assign an extension to be used in game play. The *Room Variables* that are being set are the player slots (1 through 4) and the game room creator assumes the first slot. Subsequent players will assume an empty slot once they have joined the room.

Once the request is sent and the room is created by the server, the player is automatically moved from the *Lobby* into the game room.

The game name will then appear in the game list in the *Lobby* for other players to join.

The pre-game screen provides a view of which players are currently in the game room and what their status is. In a game room, status can either be *Selecting*, *Waiting*, or *Ready*. Once all players are *Ready*, game play commences.

After spacecraft selection, player's spacecrafts are placed on the track in their start positions.

As with status updates being broadcast to the *Lobby*, when a *Ready* status update is received by the server as a `USER_VARIABLE_UPDATE`, the server checks to see if all players in the game room are ready. If so, the `ReservedRoomVariables.RV_GAME_STARTED` is set to `true`. This reserved room variable is a *Global Room Variable*, meaning, it's visible from anywhere in the *Zone*. Typically, *Room Variables* are only accessible from within a *Room*, however, when set to *Global* they become visible from any *Room* in the *Zone*. In this scenario, we're using the built in `RV_GAME_STARTED` room variable to notify players in the *Lobby* that the game has started and no more players may join.

Next, game play commences by sending all players an extension response to initiate the countdown timer.

[JAVA *sfs2x.extensions.games.spacerace.handlers.*OnUserVarChange line 124]

```
private void checkStartGate() throws SFSVariableException
{
    List<User> userList = curRoom.getUserList();
    Iterator<User> iterator = userList.iterator();

    User u = null;
    notReady:
    // Check all user in the room
    while (iterator.hasNext())
    {
        u = iterator.next();
        boolean ready = u.getVariable(SpaceRaceZoneExtension.PLAYER_READY).getBoolValue();

        // If any user has not indicated they are ready to race then set the game start indicator
```

```
            to false and break
        if (!ready)
        {
            start = false;
            break notReady;
        }
    }

    // All users in room have indicated they are ready to race
    if (start)
    {
        // Create and broadcast a global RoomVariable to notify all users not in game room that game
           has started
        RoomVariable rv = new SFSRoomVariable(ReservedRoomVariables.RV_GAME_STARTED, true);
        rv.setGlobal(true);
        List<RoomVariable> listOfVars = new ArrayList<RoomVariable>();
        listOfVars.add(rv);

        // Update the RoomVariables
        SmartFoxServer.getInstance().getAPIManager().getSFSApi().setRoomVariables(u, curRoom,
listOfVars);

        // Broadcast, using TCP, 'cd' extension to initiate countdown timer on each client
        zoneExt.send(SpaceRaceZoneExtension.COUNTDOWN, null, curRoom.getUserList(), false);
    }
}
```

When the players in the room receive the countdown extension response, each of them set their status to *In Game* (for players in the *Lobby*), and a `Countdown()` class is instantiated. The `Countdown()` class simply adds an animation that displays a three second countdown before the race starts, by calling the `startRace()` method in the `GameController()`. This method sets a timer, registers all the track objects with the `CollisionManager()`, starts the engines for all the player's spacecrafts, and adds an `ENTER_FRAME` event to update our world.

At this point, the race is playable, and keyboard events are captured to move your `MySpacecraft()` around the track. While your spacecraft is on the track, it is going to be sending an object filled with data to the server to be broadcast to all other players.

For this to run smoothly, we've made a few decisions on how to handle the amount of data being passed to the server by players and secondly how to distribute it.

If we look in the `MySpacecraft()` class on line 115 we'll see the `drive()` method. This is the most important method of the class as it defines our spacecraft's speed and direction as well as assembling an `SFSObject` to be passed to the server through an *Extension*.

Because this method is being called on every frame - for up to 4 players, we need to be very concerned with performance both on the client side and the server side. It is here where we are going to take advantage of the speed of the UDP protocol.

We are sending our coordinates among other things to the server on every frame, so missing a frame or two is acceptable in this scenario. A UDP packet may not make it to the server, but we are willing to accept that probability in return for the speed and performance improvements over TCP.

On each frame, the class member `_update:SFSObject` is filled with a number of values and then passed to the

server as an *Extension Request*. To force the extension to send using UDP, we simply add the `Boolean` value of `true` as the fourth parameter while instantiating the new `ExtensionRequest()`. Currently with SFS2X, UDP is only available with extension requests. Other requests types are forced to use TCP.

[AS3 sfs2x.games.spacerace.gameplay.MySpacecraft line 112]

```
private function drive():void
{
    _update.putInt('t', getTimer() - _startTime);
    _update.putBoolArray('d',[_up,_down,_left,_right]);
    _update.putShortArray('s',
        [
            NumberToShort( this.x,         10, false ),
            NumberToShort( this.y,         10, false ),
            this._offsetVelX,
            this._offsetVelY,
            NumberToShort( this._speed,    10, true ),
            NumberToShort( this._radians, 1000, true ),
            NumberToShort( this._steering,100, true ),
            _gameController.myLag,
            this._currentCheckpoint
        ]);

    var request:ExtensionRequest = new ExtensionRequest(Settings.EXT_SPACECRAFT_UPDATE,
                                                        _update, _gameRoom, true);
    _sfs.send(request);
```

SmartFoxServer receives this request and handles it with our `SpacecraftUpdate()` class in Java. The `SpacecraftUpdate()` class combines all incoming `SFSObjects` from all players in the room into a single response object. That object is then passed back to all players through a task scheduler that is registered in the main class of the room extension and scheduled to be run at a fixed rate of 33 milliseconds - which is approximately the set frame rate of the AIR client.

[JAVA *sfs2x.extensions.games.spacerace.*SpaceRaceGameExtension line 84]

```
sfs = SmartFoxServer.getInstance();
taskScheduler = sfs.getTaskScheduler().scheduleAtFixedRate(new BroadcastTask(), DELAY, UPDATE_
FREQUENCY, TimeUnit.MILLISECONDS);
```

The `BroadcastTask()` method simply returns our assembled object to all players in the room.

[JAVA *sfs2x.extensions.games.spacerace.*SpaceRaceGameExtension line 92]

```
private class BroadcastTask implements Runnable
{
    public void run()
    {
        userList = getGameRoom().getUserList();

        // Only send extension responses when the race has begun
```

```
        if (obj.size() > 0 && (!getParentRoom().getVariable(ReservedRoomVariables.RV_GAME_STARTED).
isNull()))
        {
            // Add the offset from game creation to now to the object being broadcast
            now = new Date();
            currentTime = now.getTime() - startTime;
            obj.putLong(TIMESTAMP, currentTime);

            // Send object containing all users latest received positions to all users in
            game room using TCP
            send(SpaceRaceGameExtension.CMD_SPACECRAFTUPDATE, obj, userList, true);

            // Cancel taskScheduler when room is empty
            if (userList.isEmpty())
            {
                taskScheduler.cancel(true);
            }
        }
    }
}
```

Once each player receives the object back, the client parses the message and loops through each player's commands and coordinates and sets the most recent commands in their respective Spacecraft classes.

Because there is always lag time in communicating with a server, the `Spacecraft()` class uses predictive positioning - it uses the latest verified position (X, Y, Rotation, Velocity, etc) of the spacecraft and calculates where it should be based on the frame the game is currently in. Players will have differing connections to the server and therefore will also have different lag times for communicating to the server.

UDP is a fast protocol, however packets always take time to travel from one point to another - even on a LAN. For illustrative purposes, we have included a lag indicator on the dashboard of the game so players can identify lag times for their opponents. This is useful to understand why some players will appear to "jump" or seem "jerky" when moving. What is happening, is that the verified coordinates are received further in the past and our predictions are more likely to be incorrect.

For example, if two players are playing on a LAN and have a 30 millisecond lag, their position on each others screens will be verified 60 milliseconds in the past (30 ms for the server to receive the message plus 30 ms for player 2 to receive the message). So the total lag for each of these players is 60 ms and we can verify their position at 60 ms in the past. We can then adjust our predictions based on that verification.

If a third player is in the game with a 100 ms lag, it will be a total of 130 ms for their verification to be received by the other players. In this case, we've predicted more than twice the amount of time for other players, and may have a "larger" adjustment to make when we receive the verification. This is what causes "jerky" movement - the higher the lag (slower the connection), the larger the time span is between verification time and current time.

Our client receives the player objects via the `SFSEvent.EXTENSION_RESPONSE` event. Again, we need to loop through the players, ignore our own coordinates, and pass other players coordinates to their `Spacecraft()` classes.

[AS3 sfs2x.games.spacerace.GameController line 1196]

```
private function onExtensionResp($e:SFSEvent):void
{
    var obj         :SFSObject    = $e.params.params as SFSObject,
        cmd         :String       = $e.params.cmd,
        playerName  :String;

    // If the command is a spacecraft update apply the information to the spacecrafts
    if(cmd == EXT_SPACECRAFT_UPDATE)
    {
        var s       :Number       = _myCraft.speed,
            num     :int          = _playerNameList.length,
            sc      :Spacecraft;

        _updateObj = obj;

        _rankList.sortOn('checkPoint', Array.NUMERIC | Array.DESCENDING);

        for(var i:int = 0; i < num; ++i)
        {
            playerName     = _rankList[i].playerName;
            sc             = _playerObjects[playerName];

            if(sc == null) continue;

            var mostRecentServerUpdate  :ISFSObject = _updateObj.getSFSObject(playerName),
                t                       :int;

            _lastCmdReceivedAt = getTimer();

            if(_updateObj.size() > 0)
            {
                if(playerName != _myName)
                {
                    sc.mostRecentCommand = mostRecentServerUpdate;
                }
            }
        }
    }
...
```

Once the Spacecraft class receives the most recent coordinates from the server, we can use those coordinates to compare our predictions with our actual placement with the `simulateFrames()` and `simulateNextFrame()` methods.

The calculations and logic used here is beyond the scope of this tutorial, however, it illustrates the ability for SmartFoxServer to receive and distribute messages for player positioning. With the use of UDP, we're able to send and receive messages multiple times a second minimizing the chance for predictive errors.

Again, if a single UDP message collides with another and some data is lost, we can easily recover with the information received in the next message.

Other extension messages use the TCP protocol to insure receipt. Events like players driving off track need to be received by other players. We cannot afford to have that type of message dropped.

Conclusion

The SFS2X Space Race includes many more gaming concept that are beyond the scope of this document. Basic interactions with the server like Public and Private messaging have been included in the application for illustrative purposes and there are other sources of more detailed information regarding that type of functionality. The goal here, was to showcase the use of the UDP protocol vs the TCP for server extensions in a real world scenario.

There is still more that could be added to this race game. For example, track selection and leveling could be added - it is the reason that the `ITrack()` interface exists. Verification for collisions, weapons, and advantages could be done on the server side to reduce cheating attempts. High scores could be kept in a database with player information.

What this game illustrates is the power, flexibility, and possibilities of the SmartFoxServer 2X framework. With the many features, APIs, expendability, and built in functionality available to developers, SmartFoxServer 2X is a superb choice for multi-player casual and hard core gaming environments and applications.

Notice

Contributors

Produced By: A51 Integrated, Toronto, Ontario, http://a51integrated.com
ActionScript Development: Wayne Helman, Fabricio Medeiros, Dave Irons, Ryan Albrecht
Java Development: Andy Rohan, Wayne Helman
Graphic Design: Steve Schoger
Project Management: Sue Timmins
Testing & QA: Reena Chohan
Documentation: Wayne Helman