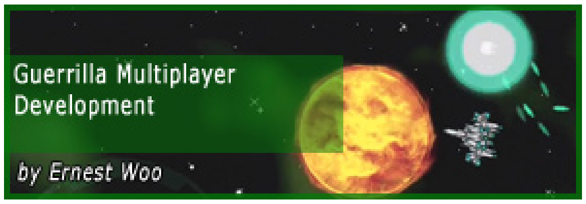
Guerrilla Multiplayer Development

By Ernest Woo



What's the best way to get a networked game up and running on a smartphone? For his space battle game ErnCon, Ernest Woo investigated existing network techniques and integrated them together, coming up with a solution useful to iOS and Android developers, which he shares within.

One of the greatest benefits of being an indie game developer is the freedom to create the exact game you want to create. Without the bureaucracy of an entire company to discourage ideas, you're free to try whatever crazy idea might work -- even if you have no clue what you're doing!

Based on user feedback from my first Android game *FRG,* and with inspiration from early multiplayer Android games like *Project INF*, I wanted the follow-up to be a mobile real time multiplayer shoot 'em-up playable over 3G. The problem was that **I had no idea how to make a multiplayer game**.

Ultimately I overcame this stumbling block and managed to release *ErnCon*. Along the way, I learned many of the technical details involved in creating a multiplayer game. Hopefully sharing my experiences will shed light on how a mobile real-time multiplayer game is developed.

Where Do I Start?

I am a Java developer by trade, with experience developing Java web applications and native applications (J2ME, Android, and iOS). I've always implemented communication between web-apps and native-apps with HTTP -- not a suitable technology for real-time games! I focused my initial research on finding answers to key concerns I had regarding developing a multiplayer game:

1. Network Address Translation (NAT). Because *ErnCon* must be playable over 3G, I had to figure out how to traverse Symmetric NAT typical of cellphone carriers. Symmetric NAT works by mapping an internal address/port (your phone) to a different external address/port (your carrier/ISP's NAT) for *every single* destination address/port (the game server) connection made.

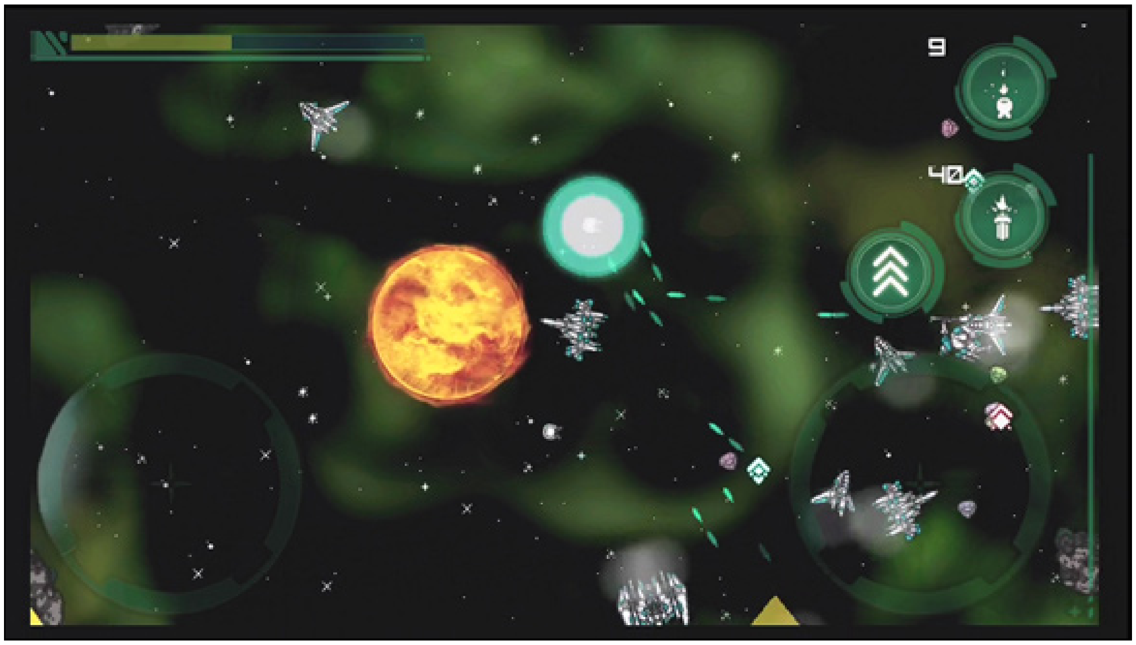
Games running on a DSL or cable modem connection behind a consumer-grade router use techniques to traverse the more permissive NAT types (Full-cone, Restricted Cone, and Port-restricted Cone), none of which really apply to Symmetric NAT.

After researching UDP Hole Punching and other NAT traversal techniques, I settled on the simpler solution of a client-server architecture where the servers always have a public IP address, i.e. the servers are not NAT'ed themselves. No special NAT traversal tricks are required for communications to a public server. For more information, read the Wikipedia article on NAT.

2. Simple network model. Without experience developing a real-time multiplayer game, I sought information on how other games implemented their networking layer. Epic Games published information on *Unreal Tournament's* network architecture, although I found it to be tied too tightly to concepts from the Unreal Engine -- I did not want to implement my own version of Unreal Engine 3. *Quake III's* network model, which I ultimately chose, proved to be much easier to understand, with its simpler notion of sending entire game-state updates delta compressed from the last known received state.

3. Reduced chance of exploits. As an indie developer, I don't have the resources to implement robust cheat detection. The architecture of the game itself must be resilient to hackers, as any data controlled by the client is ripe for exploit. A client-server architecture satisfies this concern by making the server the sole authority for all game data**.**

4. Worldwide server deployment. A real-time multiplayer game using a client-server architecture will not run well if all players in the world are connecting to servers located in a single location. To reduce lag and improve the game-play experience, servers must be strategically provisioned around the world. Players have to be segregated into regions so that geographically close players are always playing on the same servers. Amazon Web Services (AWS) was a boon for me allowing me to provision servers in various places including the US, Europe, Southeast Asia, and South America.



Implementation Details

As the saying goes, “the devil is in the details.” The effort involved in researching a proper architecture, deciding on a networking model, and settling on a provider pales in comparison to actually *implementing* everything. Without going too far into boring detail, here are some of the hurdles I've cleared:

1. Start with an existing game. Although I wanted to immediately start on *ErnCon*, which would take full advantage of multiplayer, I decided to implement a multiplayer prototype in my existing game, *FRG*. This allowed me to focus work on just the networking code and client-server development. Only when I was satisfied with the quality of *FRG*'s multiplayer implementation would I incrementally develop a new game.

2. Use and abuse Java. Because Android apps can be developed in Java, I simplified server development by *also* using Java. Doing so allowed me to quickly create a web application running in Tomcat that could run the game headless (without graphics and sound). Sharing code between client and server greatly reduced the amount of effort needed to ensure that both were simulating the same game.

---

3. Use and abuse Java even more. To rapidly implement the *Quake III* network model, I made extensive use of Java's runtime annotations and reflection. Annotations were used in game-entity classes to mark fields that could be passed over the network. The client and server assemble UDP packets based on a game-entity's list of annotated fields. Because code is shared between client and server, I only needed to make network protocol changes in one place.

4. Matchmaker service. An extra service is needed to group players into games, communicate with the game servers, and keep track of player stats like XP, cash, and inventory. Amazon Web Services provided everything I needed to develop and deploy the Matchmaker Service including:

- **EC2 load balancer**
- **Elasticache and Relational Database Service (RDS)** for the persistence layer (essentially Memcached and MySQL managed by Amazon). From memcached.org: Memcached is a "free & open source, high-performance, distributed memory object caching system."
- **Simple Storage Service (S3)** to host game assets, like item images seen in the *ErnCon* Store, and the server WAR and JAR files for the matchmaker.

5. Easy to deploy servers. Although Amazon provides many tools to help develop and deploy the *ErnCon* backend, I still had to do a lot of work to ensure that provisioning new servers is as painless as possible.

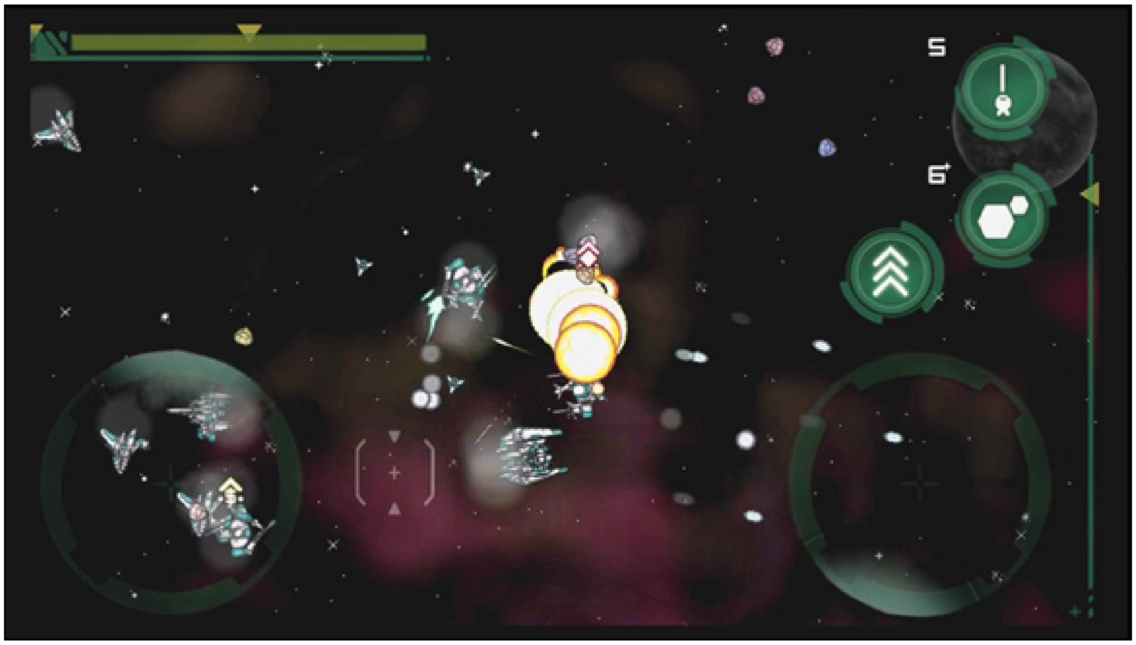
Currently I have a custom Amazon Machine Image (AMI) that on startup fetches the latest matchmaker service WAR from S3. Game servers do the same thing, except they fetch the latest game server WAR.

Game servers automatically register themselves with the matchmaker, so the matchmaker can start creating games and sending players to the game servers. When updating server code, my deployment procedure is simply:

1. Upload updated matchmaker and game server WAR and JAR files to S3.
2. Run the EC2 Instance Launch Wizard to create a new EC2 instance using my custom AMI.
3. Wait for the instance to start and verify that it is accepting players and games.
4. Repeat for all geographic regions.

Overall, provisioning a new game server takes less than five minutes.

The multiplayer architecture of *ErnCon* is something I'm proud of especially since most of it was developed in my spare time. Next step: network protocol.



3G Considerations

When designing *ErnCon's* multiplayer architecture, a major requirement was that the game be playable over 3G -- Wi-Fi should *not* be required. Playing a game should be as frictionless as possible -- *ErnCon* cannot ask a player to exit the game just to ensure he is connected to a wi-fi hotspot. Also, forcing a player to be tethered to a home wi-fi connection is the antithesis of mobile gaming! To get *ErnCon* playable on 3G, I had to consider the following throughout development:

1. Playable with typical 3G data rates. Although a concrete minimum and average data rate is not specified for 3G, research suggests a minimum 384 Kbit/s the data rate.

2. Playable with typical 3G latency. Hard data on typical latency is difficult to find. Empirical testing on T-Mobile and Verizon using a ping app showed 300ms round trip with occasional spikes to 500ms or more seemed to agree with information I could find digging through forums and articles. 300ms isn't good but with clever client-side prediction, could still be playable.

The Quake III Networking Model

Please familiarize yourself with the *Quake III* Networking Model before continuing with this article. I assume you are already familiar with concepts such as UDP networking. Some of the key points of the *Quake III* Networking Model are:

1. Server tracks what data was sent to whom. The server does a lot of work tracking what gamestate has been sent to every client. This allows the server to save bandwidth by delta compressing data.

2. Delta Compression of data. This is a fancy way of saying the server will only send a client data that has changed from the last acknowledged state. For example, if I keep track of a spaceship's x,y coordinates and facing, if the player is just moving up (along the y axis) the server only needs to send packets containing updates for that player's y coordinate.

3. Implicit handling of lag. Lag spikes or other UDP data-loss is handled by the server continuing to pump out network updates based on the last acknowledged state received by a client. This removes the need for a separate reliable acknowledgement channel.

Serializing Game Entities

Before writing any network code, I needed a way to determine what data in my existing game entities (spaceships, bullets, obstacles, etc.) should be serialized and sent over the network. As mentioned in my previous article, I opted to use runtime Java annotations to mark these "interesting" fields. The network code can scan a class definition at runtime and create a serializer used to serialize and deserialize data to be sent and received over the network.

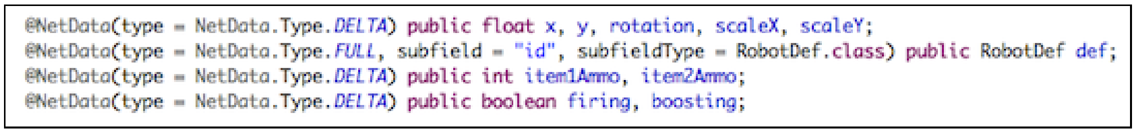
The annotations are simple, just two were needed to mark a field in different ways:

1. NetData DELTA. These fields are sent over the network whenever the field's value is different from the client's last acknowledged state. Typical DELTA fields are a game entity's position (x,y coordinates), facing, and life.

2. NetData FULL. These fields are only sent when a game entity is known to be new to a given client. Remember as mentioned above, the server tracks all data that is sent and has been sent to a client.

A FULL field is never changes its value during the life of the game entity so is never present in a gamestate update from the server after the initial payload. Typical FULL fields are a game entity's object type (what type of bullet, spaceship, or obstacle this entity represents).

The following code shows the annotated fields of the Robot class in *ErnCon,* which represents player and AI-controlled vehicles:

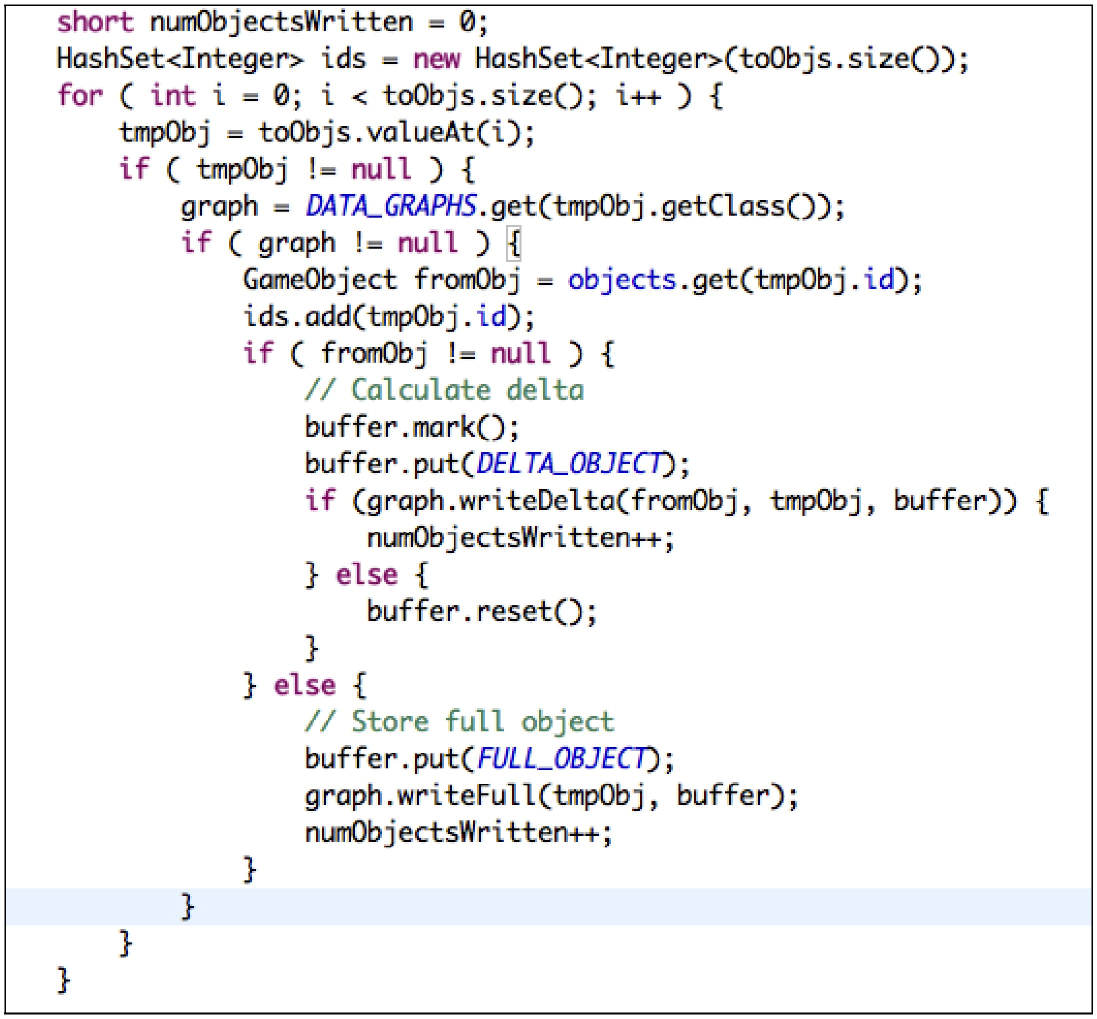


```
@NetData(type = NetData.Type.DELTA) public float x, y, rotation, scaleX, scaleY;
@NetData(type = NetData.Type.FULL, subfield = "id", subfieldType = RobotDef.class) public RobotDef def;
@NetData(type = NetData.Type.DELTA) public int item1Ammo, item2Ammo;
@NetData(type = NetData.Type.DELTA) public boolean firing, boosting;
```

Note the “def” field is a FULL field. The extra attributes on the FULL annotation are used to tell the serializer that we are serializing a non-primitive field and which field of that non-primitive field should be sent over the network.

Java reflection allows inspection of a class' field definitions for runtime annotations. Reflection also allows these field definitions to be stored in the serializer to be used to get and set values on instances of the respective game entities. The serializer, in turn, knows how to take two snapshots of a game entity and write out the differences (if any). The output is sent over the network to clients. The serializer also knows how to apply that data received from the network to an arbitrary instance making it match server-side state.

The following code snippet shows the iteration over each game entity in toObjs, finding their serializer stored in DATA_GRAPHS, and writing either a delta object from the last acknowledged game state or a full object if the object is new.



```
short numObjectsWritten = 0;
HashSet<Integer> ids = new HashSet<Integer>(toObjs.size());
for ( int i = 0; i < toObjs.size(); i++ ) {
    tmpObj = toObjs.valueAt(i);
    if ( tmpObj != null ) {
        graph = DATA_GRAPHS.get(tmpObj.getClass());
        if ( graph != null ) {
            GameObject fromObj = objects.get(tmpObj.id);
            ids.add(tmpObj.id);
            if ( fromObj != null ) {
                // Calculate delta
                buffer.mark();
                buffer.put(DELTA_OBJECT);
                if (graph.writeDelta(fromObj, tmpObj, buffer)) {
                    numObjectsWritten++;
                } else {
                    buffer.reset();
                }
            } else {
                // Store full object
                buffer.put(FULL_OBJECT);
                graph.writeFull(tmpObj, buffer);
                numObjectsWritten++;
            }
        }
    }
}
```

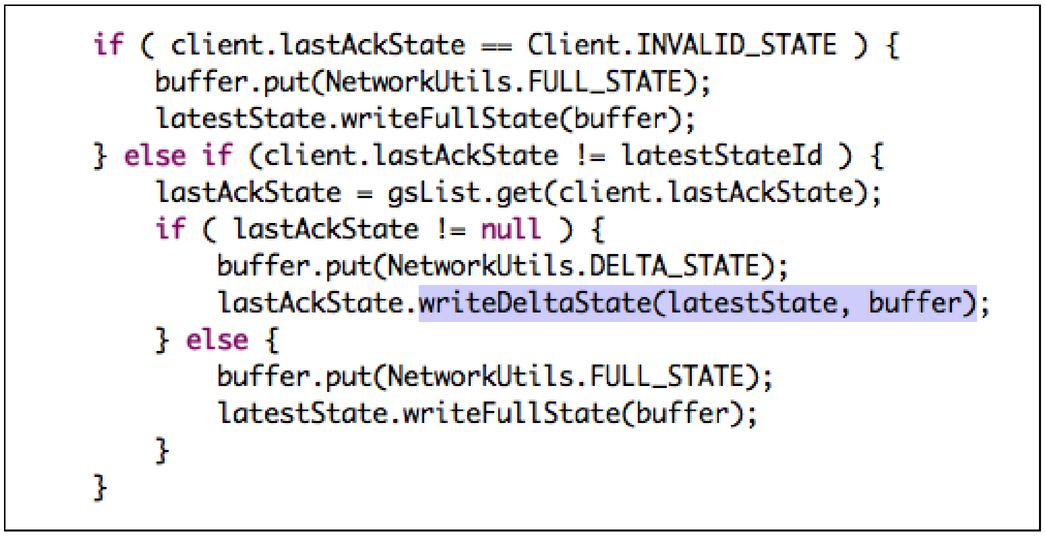
In the upcoming iOS version of *ErnCon*, I wrote an exporter that dumps all serializer and field annotation information into a JSON file that is built into the project.

Recording and Sending Game State

With serializers that know how to take individual game entities and write out the differences between two states, deltas of entire game states can be written to the network. Storing game state is very simple -- all active game entities for a given state are cloned and stored in a list managed by a GameState object. These GameState objects are used exactly as described in the *Quake III* Networking Model:

1. GameStates are saved up to the earliest GameState acknowledged by all clients.

2. For each client, create a packet of data by calculating the delta between the current GameState and the client's last acknowledged GameState as shown by the following code snippet:



```
if ( client.lastAckState == Client.INVALID_STATE ) {
    buffer.put(NetworkUtils.FULL_STATE);
    latestState.writeFullState(buffer);
} else if (client.lastAckState != latestStateId ) {
    lastAckState = gsList.get(client.lastAckState);
    if ( lastAckState != null ) {
        buffer.put(NetworkUtils.DELTA_STATE);
        lastAckState.writeDeltaState(latestState, buffer);
    } else {
        buffer.put(NetworkUtils.FULL_STATE);
        latestState.writeFullState(buffer);
    }
}
```

Note that one of the disadvantages of the *Quake III* Networking Model becomes apparent at this point: storing copies of all relevant game entities for an arbitrary number of game states can become expensive. For *ErnCon* this is not immediately a big deal because of the following:

1. Maximum number of players restricted to eight to reduce the amount of state to save (and bandwidth when communicating with clients).

2. Game servers run on EC2 instances allowing easy provisioning of new servers if needed. Multiple EC2 instances responsible for hosting games spread the load of active games.

3. Recording game states periodically as opposed to every frame. *ErnCon* only creates a new GameState object once every 50 milliseconds to conserve memory. This counts as a premature optimization and is one of the things I will start adjusting in the future.

Handling Network Data

Serializing and sending data over the network is important but not the only part of the network code in *ErnCon*. After implementing the *Quake III* Networking Model, most of my development efforts were spent on determining what data to send and how to handle it. Although this varies greatly between types of games, there are some general cases *ErnCon* handles that you should know about:

1. Client runs the simulation locally but uses network data to guide the game. Dead-reckoning in *ErnCon* simply means allowing the game simulation to run on the client in absence of data from the server. AI logic for computer-controlled enemies is also run client-side since AI behavior is deterministic for any given game-state. Important events like damage are not applied client-side although the explosion animation of a bullet hitting a spaceship is shown.

2. Ping time between client and server. For client-side prediction like dead-reckoning to work, the client must know how much time a packet takes to reach the server. Each *ErnCon* client sends a ping packet once per second. Upon receiving the ping packet, the server responds immediately. Each ping packet has an ID unique to the sending client, which is used to determine the roundtrip time. A running average of the last 3 pings is used as the basis for all dead-reckoning calculations used. In my experience, tracking more pings than 3 allows occasional network hiccups to negatively affect dead-reckoning.

3. Merging network data with client simulation. When a client receives a game state update from the server, it must go through the last acknowledged state and apply the delta-compressed data to generate a new state. This new state, in turn, must be applied to the actual local simulation. Each game entity in *ErnCon* knows how to merge data from a new state triggering appropriate animations and nudging the local entity towards the new server position based on ping time. No fancy formulas are used to interpolate between local position and updated server position -- a vector in the direction of the new position is simply applied to the object; if the object deviates from its server position for too long, then the client snaps the object to the latest server position.

4. Creating new game entities from network data. Network updates from the server can contain data on objects not known by the client -- for example, when a bullet is fired. Clients know to create new local instances of these unknown game entities.

Enough to Keep You Busy

Although I've managed to discuss the major points of *ErnCon's* multiplayer implementation, there is still much more that can be discussed. Rather than boring you with a 100-page dissertation, feel free to continue the discussion by asking questions in the comments below or via e-mail: contact@woogames.com. I'll do my best to provide code/pseudo-code to answer technical questions.

Return to the full version of this article
Copyright © 2014 UBM Tech, All rights reserved