Gama Network Presents:

Gamasutra.com

GDC 2001: 1500 Archers on a 28.8: Network Programming in *Age of Empires* and Beyond

By Paul Bettner and Mark Terrano

Gamasutra
March 22, 2001
URL:
http://www.gamasutra.com/features/20010322/terrano_01.htm

This paper explains the design architecture, implementation, and some of the lessons learned creating the multiplayer (networking) code for the *Age of Empires 1 & 2* games; and discusses the current and future networking approaches used by Ensemble Studios in its game engines.

Age of Empires Multiplayer: Design Goals

When the multiplayer code for *Age of Empires*™ was started in early 1996 there were some very specific goals that had to be met to deliver the kind of game experience we had in mind.

- Sweeping epic historical battles with a great variety of units
- Support for 8 players in multiplayer
- Insure a smooth simulation over LAN, modem-to-modem, and the Internet
- Support a target platform of: 16MB Pentium 90 with a 28.8 modem
- The communications system had to work with existing (Genie) engine
- Target consistent frame rate of 15fps on the minimum machine config

The Genie Engine was already running and the game simulation was shaping up into a compelling experience in single player. The Genie Engine is a 2D single-threaded (game loop) engine. Sprites are rendered in 256 colors in a tile-based world. Randomly-generated maps were filled with thousands of objects, from trees that could be chopped down to leaping gazelles. The rough breakdown (post optimization) of processing tasks for the engine was: 30% graphic rendering, 30% AI and Pathing, and 30% running the simulation & maintenance.

At a fairly early stage, the engine was reasonably stable -- and multiplayer communications needed to work with the existing code without substantial recoding of the existing (working) architecture.

To complicate matters further, the time to complete each simulation step varied greatly: the rendering time changed if the user was watching units, scrolling, or sitting over unexplored terrain, and large paths or strategic planning by the AI made the game turn fluctuate fairly wildly by as much as 200 msec.

A few quick calculations would show that passing even a small set of data about the units, and attempting to update it in real time would severely limit the number of units and objects we could have interacting with the player. Just passing X and Y coordinates, status, action, facing and damage would have limited us to 250 moving units in the game at the most.

We wanted to devastate a Greek city with catapults, archers, and warriors on one side while it was being besieged from the sea with triremes. Clearly, another approach was needed.

Simultaneous Simulations

Rather than passing the status of each unit in the game, the expectation was to run the exact same simulation on each machine, passing each an identical set of commands that were issued by the users at the same time. The PCs would basically synchronize their game watches in best war-movie tradition, allow players to issue commands, and then execute in exactly the same way at the same time and have identical games.

This tricky synchronization was difficult to get running initially, but did yield some surprising benefits in other areas.

Improving on the Basic Model

At the easiest conceptual level, achieving a simultaneous simulation seems fairly straightforward. For some games, using lock-step simulations and fixed game timings might even be feasible.

Since the problem of moving hundreds or thousands of objects simultaneously was taken care of by this approach -- the solution still had to be viable on the Internet with latency swings of 20 to 1,000 milliseconds, and handle changes in frame processing time.

Sending out the player commands, acknowledging all messages, and then processing them before going on to the next turn was going to be a gameplay nightmare of stop-start or slow command turnover. A scheme to continue processing the game while waiting for communications to happen in the background was needed.

Mark used a system of tagging commands to be executed two "communications turns" in the future (Comm. turns were separated in *AoE* from actual rendering frames).

So commands issued during turn 1000 would be scheduled for execution during turn 1002 (see Figure 1). On turn 1001 commands that were issued on turn 0999 would be executed. This allowed messages to be received, acknowledged, and ready to process while the game was still animating and running the simulation.

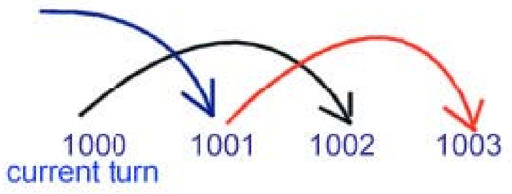

Figure 1. Tagging commands to be executed two "communications turns" in the future.

Turns were typically 200 msec in length, with commands being sent out during the turn. After 200 msec, the turn was cut off and the next turn was started. At any point during the game, commands were being processed for one turn, received and stored for the next turn, and sent out for execution two turns in the future.

"Speed Control"

Since the simulations must always have the exact same input, the game can really only run as fast as the slowest machine can process the communications, render the turn, and send out new commands. Speed Control is what we called the system to change the length of the turn to keep the animation and gameplay smooth over changing conditions in communications lag and processing speed.

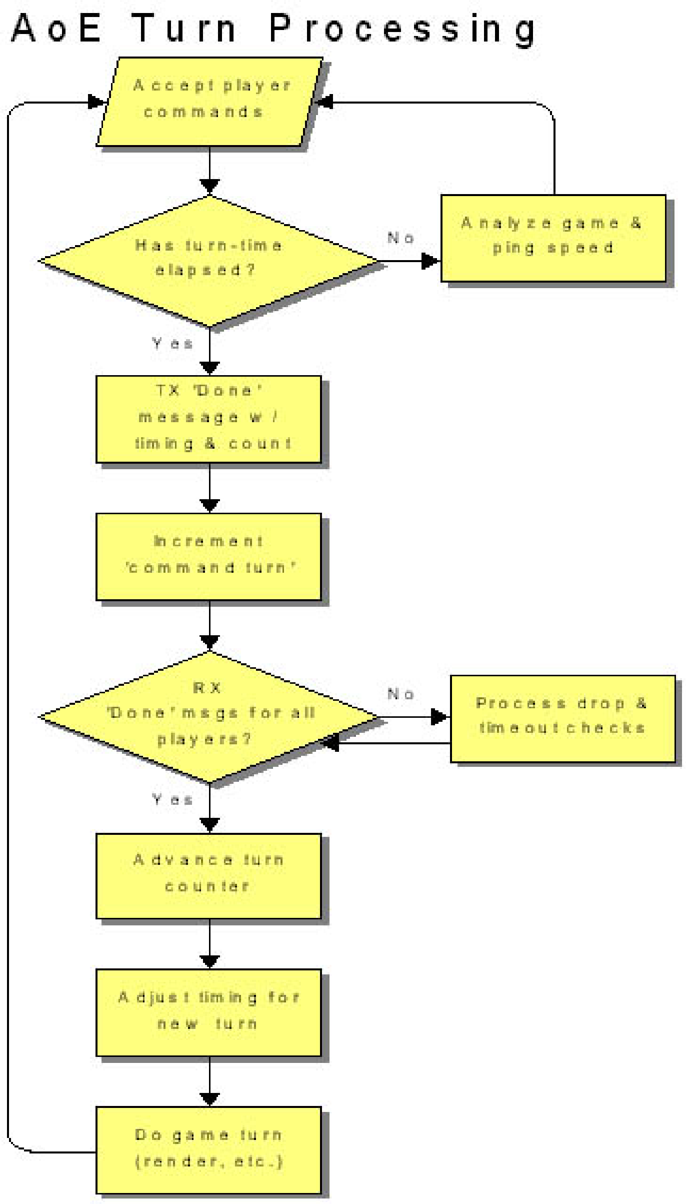
AoE Turn Processing
Accept player commands
Has turn-time elapsed?
No
Analyze game & ping speed
Yes
TX 'Done' message w / timing & count
Increment 'command turn'
RX 'Done' msgs for all players?
No
Process drop & timeout checks
Yes
Advance turn counter
Adjust timing for new turn
Do game turn (render, etc.)

Figure 2. Speed Control.

There are two factors that make the gameplay feel "laggy": If one machine's frame rate drops (or is lower than the rest) the other machines will process their commands, render all of the allocated time, and end up waiting for the next turn -- even tiny stops are immediately noticeable. Communications lag -- due to Internet latency and lost data packets would also stop the game as the players waited around for enough data to complete the turn.

Each client calculated a frame rate that it thought could be consistently maintained by averaging the processing time over a number of frames. Since this varied over the course of the game with the visible line-of-sight, number of units, map size and other factors -- it was sent with each "Turn Done" message.

Each client would also measure a round trip "ping time" periodically from it to the other clients. It would also send the longest average ping time it was seeing to any of the clients with the "Turn Done" message. (Total of 2 bytes was used for speed control.)

Each turn the designated host would analyze the "done" messages, figure out a target frame rate and adjustment for Internet latency. The host would then send out a new frame rate and communications turn length to be used. Figures 3 through 5 show how the communications turn was broken up for the different conditions.

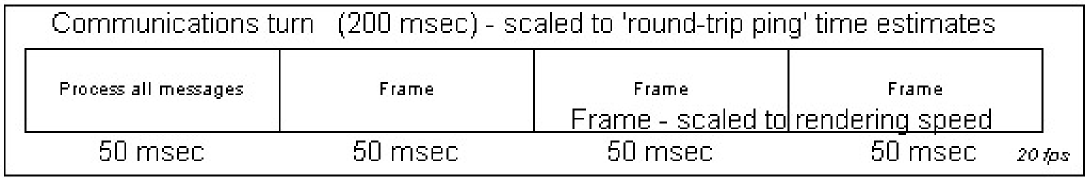
Communications turn (200 msec) - scaled to 'round-trip ping' time estimates
Process all messages
Frame
Frame
Frame
Frame - scaled to rendering speed
50 msec
50 msec
50 msec
50 msec
20 fps

Figure 3. A single communication turn.

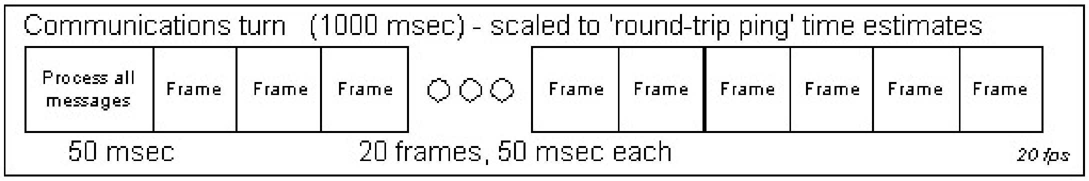

Figure 4. High Internet latency with normal machine performance.

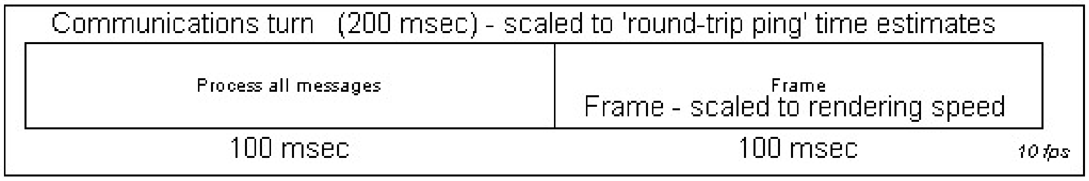
Communications turn (200 msec) - scaled to 'round-trip ping' time estimates
Process all messages
Frame
Frame - scaled to rendering speed
100 msec
100 msec
10 fps

Figure 5. Poor machine performance with normal latency.

The "communications turn" which was roughly the round-trip ping time for a message, was divided up into the number of simulation frames that on average could be done by the slowest machine in that period.

The communications turn length was weighted so it would quickly rise to handle Internet latency changes, and slowly settle back down to the best average speed that could be consistently maintained. The game would tend to pause or slow only at the very worst spikes- command latency would go up but would stay smooth (adjusting only a few milliseconds per turn) as the game adjusted back down to best possible speed. This gave the smoothest play experience possible while still adjusting to changing conditions.

Guaranteed Delivery

At the network layer UDP was used, with command ordering, drop detection and resending being handled by each client. Each message used a couple of bytes to identify the turn that execution was scheduled and the sequence number for the message. If a message was received for a past turn, it was discarded, and incoming messages were stored for execution. Because of the nature of UDP, Mark's assumption for message receipt was that "When in doubt, assume it dropped." If messages were received out of order, the receiver immediately sent out re-send requests for the dropped messages. If an acknowledgement was later than predicted, the sender would just resend without being asked anticipating the message had been dropped.

Hidden Benefits

Because the game's outcome depended on all of the users executing exactly the same simulation, it was extremely difficult to hack a client (or client communication stream) and cheat. Any simulation that ran differently was tagged as "out of sync" and the game stopped. Cheating to reveal information locally was still possible, but these few leaks were relatively easy to secure in subsequent patches and revisions. Security was a huge win.

Hidden Problems

At first take it might seem that getting two pieces of identical code to run the same should be fairly easy and straightforward -- not so. The Microsoft product manager, Tim Znamenacek, told Mark early on, "In every project, there is one stubborn bug that goes all the way to the wire -- I think out-of-sync is going to be it." He was right. The difficulty with finding out-of-sync errors is that very subtle differences would multiply over time. A deer slightly out of alignment when the random map was created would forage slightly differently -- and minutes later a villager would path a tiny bit off, or miss with his spear and take home no meat. So what showed up as a checksum difference as different food amounts had a cause that was sometimes puzzling to trace back to the original cause.

As much as we check-summed the world, the objects, the pathfinding, targeting and every other system -- it seemed that there was always one more thing that slipped just under the radar. Giant (50MB) message traces and world object dumps to sift through made the problem even more difficult. Part of the difficulty was conceptual -- programmers were not used to having to write code that used the same number of calls to random within the simulation (yes, the random numbers were seeded and synchronized as well).

Lessons Learned

A few key lessons were learned in the development of the networking for *Age of Empires* that are applicable to development of any game's multiplayer system.

Know your user. Studying the user is key to understanding what their expectations are for multiplayer performance, perceived lag, and command latency. Each game genre is different, and you need to understand what is right for your specific gameplay and controls.

Early in the development process Mark sat down with the lead designer and prototyped communications latency (this was something that was revisited throughout the development process). Since the single-player game was running, it was easy to simulate different ranges of command latency and get player feedback on when it felt right, sluggish, jerky, or just horrible.

For RTS games, 250 milliseconds of command latency was not even noticed -- between 250 and 500 msec was very playable, and beyond 500 it started to be noticeable. It was also interesting to note that players developed a "game pace" and a mental expectation of the lag between when they clicked and when their unit responded. A consistent slower response was better than alternating between fast and slow command latency (say between 80 and 500 msec) -- in that case a consistent 500 msec command latency was playable, but one that varied was considered "jerky" and hard to use.

In real terms this directed a lot of the programming efforts at smoothness -- it was better to pick a longer turn length and be certain that everything stayed smooth and consistent than to run as quickly as possible with occasional slow-downs. Any changes to speed had to be gradual and in as small increments as possible.

We also metered the users demands on the system -- they would typically issue commands (move, attack, chop trees) averaging about every 1.5 to 2 seconds, with occasional spikes of 3 to 4 commands per second during heated battles. Since our game built to crescendos of frantic activity the heaviest communications demands were middle and late game.

When you take the time to study your user behavior you'll notice other things about how they play the game that can help your network play. In *AoE*, clicking repeatedly when the users were excitedly attacking (clik-lik-lik-lik-lik -- go go go) was causing huge spikes in the number of commands issued per second -- and if they were pathing a large group of units -- huge spikes in the network demand as well. A simple filter to discard repeat commands at the same location drastically reduced the impact of this behavior.

In summary, goals of user observation will let you:

- Know the latency expectations of the user for your game
- Prototype multiplayer aspects of play early
- Watch for behavior that hurts multiplayer performance.

Metering is king. You will discover surprising things about how your communications system is working if you put in metering early, make it readable by testers, and use it to understand what is happening under the hood of your networking engine.

Lesson: Some of the problems with AoE communication happened when Mark took the metering out too early, and did not re-verify message (length and frequency) levels after the final code was in. Undetected things like occasional AI race conditions, difficult-to-compute paths, and poorly structured command packets could cause huge performance problems in an otherwise well tuned system.

Have your system notify testers and developers when it seems like it is exceeding boundary conditions -- programmers and testers will notice during development which tasks are stressing the system and let you know early enough to do something about it.

Take the time to educate your testers in how your communications system works, and expose and explain the summary metering to them -- you might be surprised what things they notice when the networking code inevitably encounters strange failures.

In summary, your metering should:

- Be human readable and understandable by testers
- Reveal bottlenecks, slowdowns, and problems
- Be low impact and kept running all the time.

Educating the developers. Getting programmers who are used to developing single-player applications to start thinking about a detachment between the command being issued, received, and being processed is tricky. It is easy to forget that you are requesting something that might not happen, or might happen seconds after you originally issue the command. Commands have to be checked for validity both on send and receive.

With the synchronous model, programmers also had to be aware that the code must not depend on any local factor (such as having free time, special hardware, or different settings) when it was in the simulation. The code path taken on all machines must match. For example having random terrain sounds inside the simulation would cause the games to behave differently (saving and re-seeding the pseudo-random number generator with the last random number took care of things inside the simulation that we needed to be random but not change the simulation.

Other lessons. This should be common sense -- but If you depend on a third-party network (in our case DirectPlay), write an independent test application to verify that when they say "guaranteed delivery" that the messages get there, that "guaranteed packet order" truly is, and that the product does not have hidden bottlenecks or strange behaviors handling the communications for your game.

Be prepared to create simulation applications and stress test simulators. We ended up with three different minimal test applications, all to isolate and highlight problems like connection flooding, problems with simultaneous matchmaking connects, and dropped guaranteed packets.

Test with modems (and, if you are lucky, modem simulators) as early as possible in the process; continue to include modem testing (as painful as it is) throughout the development process. Because it is hard to isolate problems (is that sudden performance drop because of the ISP, the game, the communications software, the modem, the matchmaking service, or the other end?) and users really don't want to hassle with flaky dialup connections when they have been zipping along at instant-connection LAN speeds. It is vital that you assure testing is done on modem connections with the same zeal as the LAN multiplayer games.

Improvements for *Age of Empires 2*

In *Age of Empires 2: The Age of Kings*, we added new multiplayer features such as recorded games, file transfer, and persistent stat tracking on The Zone. We also refined the multiplayer systems such as DirectPlay integration and speed control to address bugs and performance issues that had come up since the release of *Age of Empires*.

The game recording feature was one of those things that you just happen to stumble upon as an "I could really use this for debugging" task that ends up as a full-blown game feature. Recorded games are incredibly popular with the fan sites as it allows gamers to trade and analyze strategies, view famous battles, and review the games they played in. As a debugging tool, recorded games are invaluable. Because our simulation is deterministic, and recorded games are synchronous in the same way that multiplayer is synchronous, a game recording gave us a great way of passing around repro cases for bugs because it was guaranteed to play out the exact same way every time.

Our integration with the matchmaking system on The Zone was limited to straightforward game launching for *Age of Empires*. In *Age of Kings* we extended this to allow for launch parameter control and persistent stat reporting. While not a fully inside-out system, we utilized DirectPlay's lobby launch functionality to allow The Zone to control certain aspects of the game settings from the pre-game tables, and "lock" those settings in once the game was actually launched. This allowed users to better find the games they wanted to play in, because they could see the settings at the matchmaking level, rather than waiting to launch into the game setup screen. On the backend we implemented persistent stat reporting and tracking. We provide a common structure to The Zone, which we fill out and upload at the end of a game. The data in this structure is used to populate a number of user ratings and rankings viewable on The Zone's web site.

RTS3 Multiplayer: Goals

RTS3 is the codename for Ensemble's next-generation strategy game. The RTS3 design builds on the successful formula used in the Age of Empires series games, and calls for a number of new features and multiplayer requirements.

- Builds on the feature set of *Age of Empires 1* and *2*. Design requirements such as internet play, large diverse maps, and thousands of controllable units are a given.
- 3D -- RTS3 is a fully 3D game, with interpolated animation and non-faceted unit position and rotation.
- More players -- possible support for more than eight players.
- TCP/IP support -- 56k TCP/IP internet connection is our primary target.
- Home network support -- Support end-user home network configurations including firewalls and NAT setups.

With RTS3, we made the decision early on to go with the same underlying network model as *Age of Empires 1* and *2* -- the synchronous simulation -- because the RTS3 design played to the strengths of this architecture in the same ways. With *AOE/AOK*, we relied on DirectPlay for transport and session management services, but for RTS3 we decided to create a core network library, using only the most basic socket routines as our foundation and building from there.

The move to a fully 3D world meant that we had to be more sensitive to issues of frame-rate and overall simulation smoothness in multiplayer. However, it also meant that our simulation update times and frame-rate would be even more prone to variation, and that we would be devoting more time overall to rendering. In the Genie engine, unit rotations were faceted and animations were frame-rate locked -- with BANG! we allowed for arbitrary unit rotation and smooth animation which meant that the game would be visually much more sensitive to the effects of latency and see-sawing update rates.

Coming out of development on *Age of Kings*, we wanted to address those critical areas where more up-front design and tool-set work would give the biggest payoff in terms of debugging time. We also realized how important the iterative play-testing process was to the design of our games, and so bringing the multiplayer game online as early as possible was high priority.

RTS3 Communications Architecture

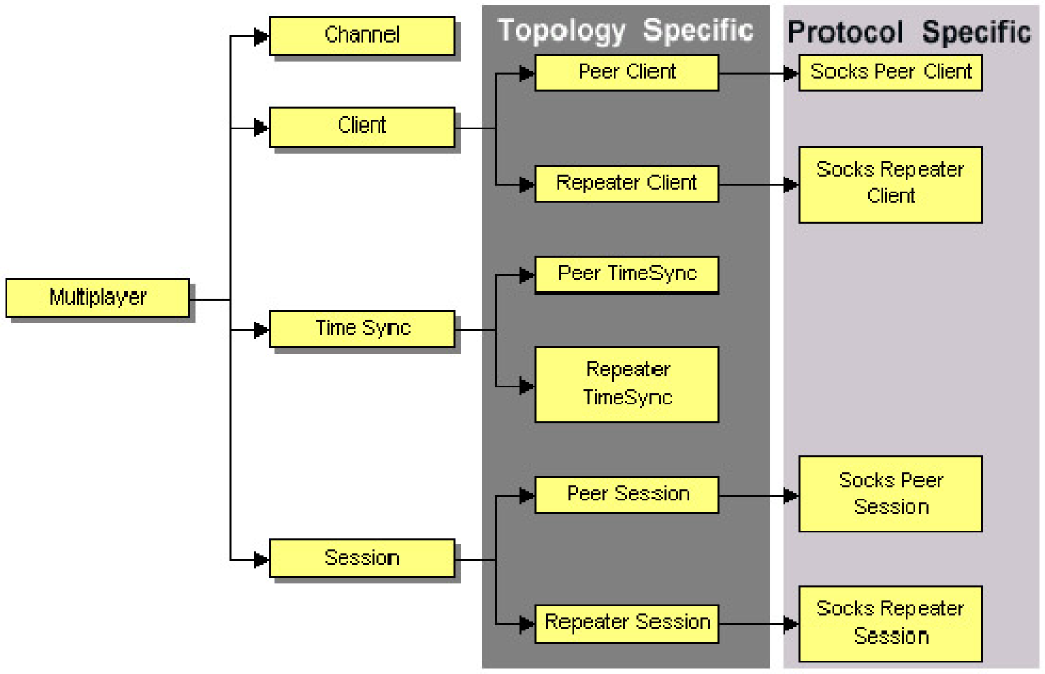

Figure 6. RTS3's strongly object-oriented network architecture.

An OO approach. RTS3's network architecture is strongly object oriented (see Figure 6). The requirements of supporting multiple network configurations really played to the strengths of OO design in abstracting out the specifics of platform, protocol, and topology behind a set of common objects and systems.

The protocol specific and topology specific versions of the network objects have as little code as possible. The bulk of the functionality for these objects has been isolated in the higher-level parent objects. To implement a new protocol, we extend only those network objects that need to have protocol specific code (such as client and session, which need to do some things different based on the protocol). None of the other objects in the system (such as Channels, TimeSync, etc.) need change because they interface with client and session only through their high level abstract interfaces.

We also employ the use of aggregation to implement multi-dimensional derivation (such as with channels, that have an ordered/non-ordered axis of derivation, as well as a peer/repeater axis of derivation) behind a single generic interface. Virtual methods are also used for non-intensive notifications, rather than using callback functions.

Peer topology. The Genie engine supported a peer-to-peer network topology, in which all clients in the session connect to all the other clients in a star configuration. With RTS3 we have continued the use this topology because of its inherent benefits when applied to the synchronous simulation model.

The peer topology implies a star configuration of connected clients in a session (Figure 7). That is, all clients connect to all other clients. This is the setup that *Age 1* and *2* utilized.

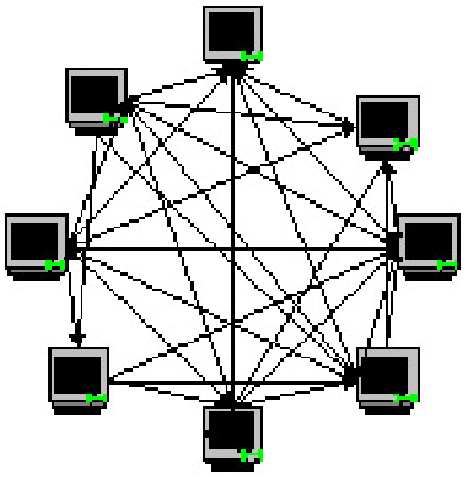

Figure 7. A "star" configuration of peer-to-peer clients in a session.

Peer-to-peer strengths:

- Reduced latency due to the direct client-client nature of the system, rather than a client-server-client roundtrip for messages.
- No central point of failure -- if a client (even the host) disconnects from the session, the game can continue.

Peer-to-peer weaknesses:

- More active connections in the system (Summation n=0 to k-1 (n)) -- means more potential failure points and latency potential.
- Impossible to support some NAT configurations with this approach.

Net.lib. Our goal when designing the RTS3 communications architecture was to create a system that was tailored for strategy games, but at the same time we wanted to build something that could be used for in-house tools and extended to support our future games. To meet this goal, we created a layered architecture that supports game-level objects such as a client and a session, but also supports lower level transport objects such as a link or a network address.

RTS3 is built upon our next-generation BANG! engine, which uses a modular architecture with component libraries such as sound, rendering, and networking. The network subsystem fits in here as a component that links with the BANG! engine (as well as various in-house tools). Our network model is divided up into four service layers that look almost, but not entirely, unlike the OSI Network Model, if you applied it to games (see Figure 8).

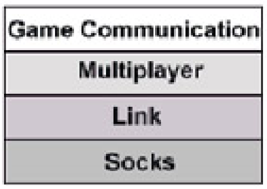

Figure 8. The four service layers of the network model.

Socks, Level 1

The first level, Socks, provides the fundamental socket level C API, and is abstracted to provide a common set of low level network routines on a variety of operating systems. The interface resembles that of Berkley sockets. The Socks level is primarily used by the higher levels of the network library, and not really intended to be used by the application code.

Link, Level 2

Level 2, the Link Level, offers transport layer services. The objects in this level, such as the Link, Listener, NetworkAddress, and Packet represent the useful elements needed to establish a connection and send some messages across it (see Figure 9).

- **Packet:** This is our fundamental message structure -- an extensible object that automatically manages its own serialization/de-serialization (via pure virtual methods) when sent across a link object.
- **Link:** a connection between two network endpoints. This can also be a loopback link, in which case the endpoints both reside on the same machine. Send and receive methods on a link know how to operate with Packets and also with void* data buffers.
- **Listener:** a link generator. This object listens for incoming connections, and spawns a link when a connection is established.
- **Data stream:** this is an arbitrary meter-able data stream across a given link -- used to implement file transfer, for example.
- **Net Address:** a protocol independent network addressing object **Ping:** a simple ping class. Reports on the network latency present in

a given link.

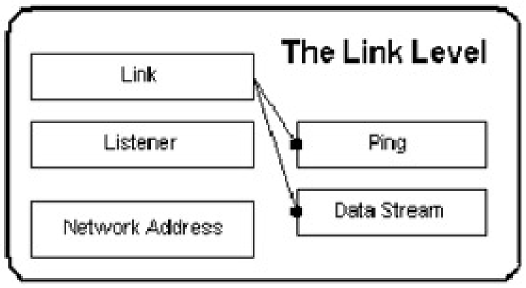

Figure 9. The Link level.

<u>Multiplayer, Level 3</u>

The multiplayer level is the highest level of objects and routines available in the net.lib API. This is the layer that RTS3 interfaces with as it collects lower level objects, such as links, into more useful concepts/objects such as clients and sessions.

The most interesting objects in the BANG! network library are probably those that live at the multiplayer level. Here, the API presents a set of objects that the game level interacts with, and yet we maintain a game-independent approach in the implementation.

- **Client:** this is the most basic abstraction of a network endpoint. This can be configured as a remote client (link) or local client (loopback link). Clients are not created directly, but are instead spawned by a session object.
- **Session:** this is the object responsible for the creation, connection negotiation, collection and management of clients. The session contains all the other multiplayer-level objects. To use this object, the application simply calls host() or join(), giving it either a local address, or remote address respectively and the session handles the rest. These responsibilities include automatically creating/deleting clients, notification of session events, and the dispatch of traffic to the appropriate objects.
- **Channel and Ordered Channel:** this object represents a virtual message conduit. Messages sent across a channel will be automatically separated out and received on the corresponding channel object on remote clients. An ordered channel works with the TimeSync object to guarantee that the ordering of messages received on that channel will be identical on all clients.
- **Shared Data:** Represents a collection of data shared across all clients. You extend this object to create specific instances that

contain your own data types, and then use the built in methods to enable the automatic and synchronous updating of these data elements across the network.

- **Time Sync:** Manages the smooth progression of synchronized network time across all clients in a session.

Game Communications, Level 4

The communications level is the RTS3 side of things. This is the main collection of systems through which the game interfaces with the network library, and it actually lives within the game code itself. The communications layer provides a plethora of useful utility functions for the creation and management of multiplayer-level network objects and attempts to boil down the game's multiplayer needs into a small easy to use interface.

New Features and Better Tools

Improved sync system. Nobody on the *Age of Empires* development team would argue the need for the best sync tools possible. As with any project, when you look back on the development process during a postmortem, some areas always stand out as the ones you spent the most time on, but could have spent much less time on given more up-front work. Synchronization debugging was probably at the top of this list as we started development on RTS3.

The RTS3 synchronization tracking system is primarily geared towards rapid turn-around on sync bugs. Our other priorities in developing it were ease of use for the developers, the ability to handle an arbitrarily massive amount of sync data pouring through the system, the ability to totally compile out synchronization code in a release build, and finally the ability to completely change our test configuration by toggling some variables rather than requiring a recompile.

Sync checking in RTS3 is done through two sets of macros:

```
#define syncRandCode(userinfo) gSync->addCodeSync(cRandSync,
userinfo, __FILE__, __LINE__)
```

```
#define syncRandData(userinfo, v)
gSync->addDataSync(cRandSync, v, userinfo, __FILE__, __LINE__)
```

(There is a set of these macros per sync "tag," where a tag represents a given system to be synced -- in this example, the random number generator, cRandSync) These macros both take a `userinfo` string parameter, which is a name or indication of the specific item being synced. For example, a sync call might look like:

```
syncRandCode("syncing the random seed", seed);
```

Synchronous console commands and config variables. Console commands and configuration variables are of immense value to the development process, as any *Quake* mod creator will attest to. Console commands are simple function calls, done via a startup configuration file, the console within the game, or UI hooks, that call into any arbitrary game functionality. Config variables are named data types, exposed through simple get, set, define and toggle functions that we use for all sorts of testing and configuration parameters.

Paul derived multiplayer-enabled versions of our console command and config variable systems. With these, we are able to easily turn a normal config variable (such as enableCheating) into a multiplayer config variable by adding a flag to the config variable's definition. With this flag enabled, that config variable will then be passed around in a multiplayer game, and synchronous game decisions (such as whether to allow free resource tributing) can be based off of the value. Multiplayer console commands is a similar concept -- calls to a multiplayer-enabled console command are passed around and executed synchronously on all client machines.

Through the application of these two tools, the developers have a simple way to use the multiplayer system without writing any lines of code. They can quickly add new testing tools or configurations, and easily enable them in the network environment.

Summation

The synchronous simulation, peer to peer model was used successfully in the Age of Empires series of games. While it is critical to acknowledge the importance of investing time creating tools and technologies to combat the key challenges of this approach (such as synchronization and network metering), the viability of this architecture is proven when applied to the real-time strategy genre. The subsequent improvements we have implemented for RTS3 lead to an overall multiplayer experience that is virtually indistinguishable from single player in all but the most horrible network conditions.

http://www.gamasutra.com/features/20010322/terrano_01.htm