An Entity System for Networked Game Development

or: Networked Games Development with realXtend Tundra SDK

This document was generated on 2012-08-14 at 10:05.

toni

jukka

jonne?

rauli?

erno? ...

Contents

Abstract

We present an inherently networked entity system for creating multiuser applications, including networked games. The entities aggregate components with attributes which are automatically synchronized in a client-server setup. So called entity-actions can be called both locally and remotely, they are a simple form of RPC. This system is implemented in the open source realXtend Tundra SDK which is a complete platform for multiuser 3d applications. The API of Tundra is evaluated critically to analyze the entity model and to identify areas for improvement in future work.

The goal is to make creating efficient multiplayer games easy, with for example just Javascript logic code, without the need to invent own network messages for simple functionality. Custom messages are supported for efficiency in more complex cases.

Besides the generic attribute sync, the Tundra SDK includes custom messages and logic for moving rigid bodies, utilizing the physics module also on the client side and by implementing (XXX basic sensible

things / nice clever tricks / best practices from the literature / something). Measurements from these optimizations are presented -- with the optimizations the maximum number of moving objects in a scene went up by N.

Introduction

Developing a networked multiplayer game is typically (unavoidably) more complex than a local game. Multiplayer logic is a part of it, compared to single player games, but a large part of the complexity comes with the networking: Synchronizing state, triggering events remotely, arbitrating possible conflicts due to simultaneous actions from different clients, dealing with problems such as dropped messages and aborted connections etc. All that work is away from what a game developer ideally would be focusing on: developing the gameplay itself.

The question here is whether and how a platform or a library can make networked game programming easier. In computer science in general, any problem can be addressed by introducing a new layer of abstraction, but there are caveats. Many layers of abstraction with different concepts may just add the cognitive load for the developer, especially if underlying issues such as network connectivity problems still propagate through to the top and need to be dealt with. What would be the best core concepts for productive development of robust networked games?

The model presented here is an entity-component model, similar to several contemporary games and engines. It uses aggregation, instead of inheritance, to compose all game entities as sets of component instances. All the scene data is in the attributes of the components, which are automatically synchronized over the network among all the participants using a client-server setup. Additionally, so-called entity-actions can be called on the entities -- either locally or over the network, it is a simple remote procedure call (RPC) mechanism. In this paper, we evaluate the model by analyzing game development done using it, and compare the approach with the alternative of just using a networking library to create connections and do messaging in own game code.

The evaluation is made using the open source Tundra SDK from the realXtend initiative. Tundra has been built from the start using this entity-component model, also for the basic core functionality, and it is the extensibility mechanism that Tundra provides for additional functionality as well. We give a brief overview of Tundra as a platform overall, but the focus is on the networking and the abstractions provided for application developers. In addition to reviewing the ease of development, we also analyze the efficiency and robustness of how the abstract entity-component model with the automatic synchronization maps to concrete network load.

(NOTE: something more is needed about how APIs and platforms can be evaluated, can qualitative research about programming environments be done and how? both here to intro and to background/related work! XXX)

The article is organized as follows: next we review background and related work from the literature. Then the design of the entity-component model is described, and illustrated with two application examples: 1. A complete treatment of the source code of a minimal networked multiplayer Pong and 2. Selected details from a complex watershed environment which hosts two interconnected multiplayer minigames. Then the underlying networking layer is analyzed to see how the higher level game codes translate to actual traffic, how much bandwidth is used (etc? XXX). (Finally ..?)

Background / related

The entity model presented here is by far not unique, quite the opposite, as the idea is to apply best practices from the literature.

We adopted the aggregation based technique after studying [ecref (gta4?)], and for example Unity3d uses components the same way, also as an extension point for plugins [unity ref?]. Typically these entity systems are not, however, inherently networked (?). One game platform that we studied when making the attribute synchronization was Syntensity, which adds a Javascript scripting system with networked attributes to the Cube FPS engine based Sauerbraten collaborative networked system.

Also simplifying networked game development by providing a platform which takes care of all the basics, such as client-server connectivity and synchronizing movements and other state, is nothing new. All FPS mods, scriptable MMOs and engines such as Unreal (and Half-Life and which are the relevant ones?) feature that.

As the target here is to analyze the quality of the model by the API it provides for networked game developers, we could in principle use some of the aforementioned proprietary platforms for the study as well. However the Tundra SDK is, by being liberally licensed open source (apache), completely available for anyone for most detailed analysis and freely modifyable to test possible improvements. Our results are easily repeatable as anyone can download and run the same codes. (Homura-HUGS paper argues for and does that too, NHUGS: Towards scalability testing for MMOGs within an extensible, open architecture).

Yet a comparative analysis with multiple platforms, including proprietary commercial ones, would certainly be fruitful (XXX -- perhaps we can do that to a limited extent here, by talking with folks with experience with Unity, Unreal etc. dev, and OTOH by reviewing things like Syntensity in more detail .. return to this in discussion/evaluation?).

- the APIs of those, the app dev model: are e.g. connections dealt with at all typically etc? how is data synched (or is it even needed in those, server logic?, scripts?). how do messaging things work (room for improvement in Tundra perhaps?)

...

Game development using the Entity-Component model

The entity-component model is an abstract design, not tied to any specific platform. It is presented here first on the conceptual level, and illustrated with examples. Finally the implementation of the model in the Tundra SDK is described, both to analyze how the design works out in a concrete platform and to identify possible improvements for the conceptual models and the implementation there.

The abstract model

The core of the entity-component model is very simple: An entity is just an identity, without any type or data (apart from the id). It is used to aggregate components, which have attributes for synchronized & persistent data, and code to implement the functionality of the component. An application is a collection if entities.

This aggregation based approach steps away from the inheritance oriented class hierarchies which were typical in games earlier, to avoid problems with deep class hierarchies and difficulties of sharing a piece of functionality across otherwise remote types in a hierarchy [ecref]. It provides a uniform way of programming a piece of functionality for all types of entities.

For example, all positioned entities in a 3d scene can have a Placeable component which contains the scene node transform (position, orientation and scale). Then any code that deals with positions just works for all kinds of entities -- lamps, cameras, players or whatever -- as the placeable component is the same in all of them, and the entities are not typed. The other functionalities of an entity are implemented in other components, for example a light of camera component, but that is independent of the placeable aspect.

Furthermore, all the component data is handled in a unified way with the generic attribute mechanism. A component specifies the attributes it contains. The generic systems then take care of synchronizing the data across the network, and of persisting it (saving to file or database). No special network messages are required to implement features, such as having coloured lights or sound sources with varying audio volume levels -- the light and sound components just define their data as attributes. Changes in attribute data are communicated with generic attribute synchronization messages which are specific for the data type (float, string, ..) but independent of the containing component.

Additionally, so-called entity-actions can be registered as callback functions in the entities. They can be called both locally and remotely and are a simple form of remote procedure calls (RPC). The entity-actions are called indirectly: the callback handlers are implemented in components, but the calls are on the entity.

That is to be able to provide a uniform interface to different but related functionality: For example, a Hide action can be registered so that a UI button or some game logic code can hide a set of entities. The details of how to hide a certain kind of an entity depends on the components it uses to display: for example whether it is a mesh, a particle system, a piece of text or some UI element. By implementing the Hide action in all the different components but routing the call via the entities the same interface works for all implementations.

By default, the entities and components are replicated and persistent, i.e. synchronized over network and saved to files. They can also be configured to be local only, either on server or client side, and/or temporary. Temporary entities are typically created by application logic code by a script in some other, persistent entity. For example the reference avatar implementation for virtual worlds like usage in Tundra SDK has logic code to create visible avatars upon new user logins, and the avatars are replicated, but they are temporary and hence not stored when the world state is saved while users are logged in.

Example 1: Pong as the Hello World of multiplayer games

Pong is a minimal multiplayer game, so let's use it as a simple example of making a networked game using the entity-component model. We are using the realXtend Tundra SDK for the evaluation here and it is a 3d scenegraph engine with rigid body physics simulations so the game environment and mechanisms are built with those.

The Pong scene consists of -- similarily to the game of tennis -- the playing field, two paddles for the players and the ball. In this example the static scene is created with a 3d modeling program (in this case Blender3d). The scene is exported from Blender to Tundra SDK, at which point it is converted to the entity-component model: all the visible entities have a Placeable component for being in the scene, Mesh for the visual geometry and Rigidbody for the physics simulation.

To make the game logic, an additional invisible entity is added, let's call it PongGame. We write the code in Javascript, for which the mechanism in Tundra is to add a Script component with a reference to the .js file as an attribute. We want to show a basic GUI in the clients to visualize the game state: whether a game is running or not, and what is the score. So let's add also a custom component with that data in attributes, PongGameState with Boolean:Running and integer attributes for player 1 and 2 scores. That way the data is automatically synchronized to clients as well so they can easily use it in the GUI code. The physics simulation bouncing the ball is ran on the server side by default, and that is where we want to have all the logic code of checking when a player scores, starting and stopping games etc.

In addition to having the logic code and the game state data, we need to handle clients / players joining and leaving the game. Joining is triggered with a GUI button in a client, which sends an entity-action called "JoinGame" to the PongGame application entity, to be handled on the server side.

The game does not need to know about clients logging into the server, as we can have any number of spectators there. As joining the game is made as a separate action, the game does not need to care when new bare client connections are established. But we need to handle disconnects when some player connection is dropped in the middle of a running game. Network connections in Tundra are outside the entity model, but hooks for dealing with them are provided in the builtin core API instead. In this case, the server api object has an event called UserDisconnected to which we can connect our handler.

We begin the game, for simplicity, when two players have joined in. They are assigned controls for their own paddles, for example the mouse y coordinate can be mapped to the corresponding position along the side of the table. We can manipulate the paddle position directly in the client by the same code which reads the mouse position. This is optimal for the control feel to avoid any lag in the visual response for the hand movements, but can be problematic when the physics are executed on the server side and there is network latency. The player can see the ball passing through her paddle, if the server did not receive the paddle movement in time. Another possibility is to communicate the controls to the server, move the paddles there, and thereby get the visual feedback in the client only after the full roundtrip. This could allow the player to compensate for the latency, but also make the controlling more difficult due to the delay. For a study of different strategies for dealing with latency in the game of pong, see [PongPaper].

The positions of all objects, the transform attributes of the placeable components in them, are synchronized automatically so all the participants get the paddle and ball positions automatically. The bouncing of the ball is handled automatically by the physics engine. The game code only needs to:

1. Start the game, when two players join, by giving the ball some initial velocity
2. Handle player controls of the paddles during the game
3. Check for the winning condition (ball passes either side) and keep score

4. Handle the user actions to join and thereby start the game, and the different cases when the game is stopped (win, user decides to stop, or connection drops).

Arguably this way to implement a networked multiplayer game of pong is very simple, and succesfully hides all the details of networking from the game developer. It is implemented in two modules, the client and the server side codes, with X hundred lines total. (e.g. the example there does a bit more manually, even though is largely similar: http://www.unionplatform.com/?page_id=1229&page=2 -- and is much larger, N>10 classes)

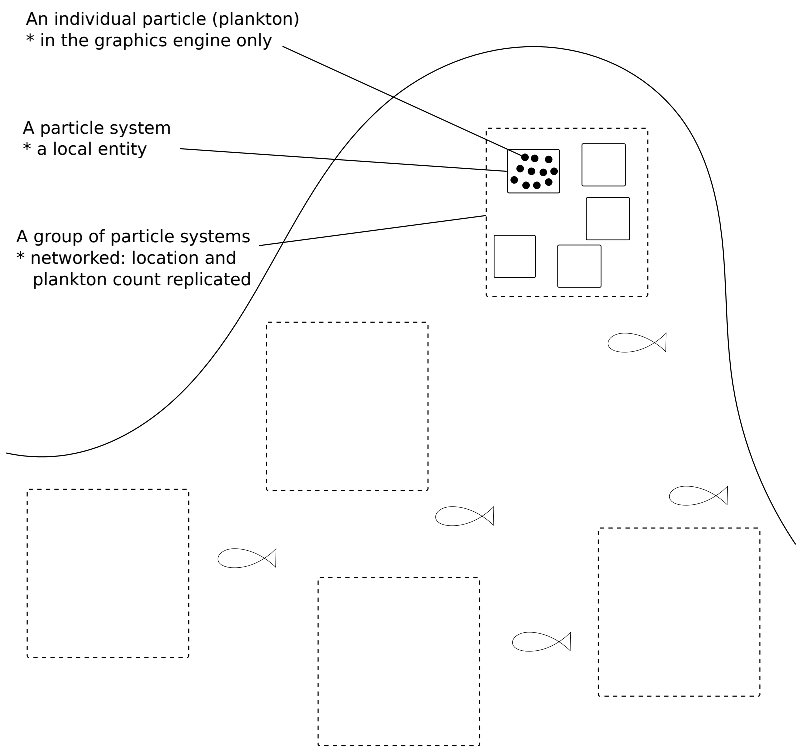
The complete source code and the required 3d assets to run this pong implementation on the Tundra runtime is available from https://github.com/realXtend/doc/tree/master/netgames/PongMultiplayer

Example 2: Swarming plankton as food for fish in the sea

A simple way to make a trivial pong implementation may be nice, but does the approach work for real, more complex games? We and others have implemented a range of applications using the entity-component model on the Tundra SDK, and this section is to analyze issues encountered with more complex functionality. The particular case is from an open source application made at the end of the original realXtend project, as a public demo of the Tundra SDK. That is the Smithsonian Latino Virtual Museum's Virtual Watershed Initiative, and in particular the experimental Anchovy game made to the sea bay there.

The whole watershed environment hosts a range of animals of different scale, from white-tailed deer and opossum to osprey, sea bass and the anchovy. The idea is that by taking the role of an animal they player (a child visiting the museum for example) can learn about biology. In the anchovy game, the player controls the little fish from a 3rd person angle, trying to find food such as plankton in the sea. The idea is to have quite a lot of little plankton clouds there, but so that when multiple players consume it the amount decreases.

To be able to render a lot of little plankton, we use particle systems. The individual particles in the particle systems move slightly at random, to give a feel of them floating around in the water. To have enough particles to fill parts of the sea bay, we easily need tens of particle systems with hundreds of particles in each. Synchronizing all those little movements would take an immense amount of bandwidth, also considering that many other things are going on in the scene as well. To cut down the traffic, not only are the individual particles local only, but also the movement of a single particle system is not communicated. Instead, we form clusters of 5 particle systems which move around as a loose group, and synchronize only the positions of such clusters. This way we can have lots of plankton in approximately the same positions for the different players. Also the amount of plankton left in a cluster is synchronized. The idea is that the different players see the plankton clouds in same areas of the sea bay, and see them diminish when eaten, but with relatively little network traffic.


An individual particle (plankton)
* in the graphics engine only
A particle system
* a local entity
A group of particle systems
* networked: location and
plankton count replicated

Composing a lot of plankton to the sea in a multiplayer game in a hieararchy of local only and networked entities

That system is implemented by having the game code (Javascript) create the particle systems in local-only entities, which are not synchronized over the network at all. Only the clusters are normal replicated Tundra entities, for which the movement synchronization works.

The fish themselves are normal replicated entities for which the server is authorative. That required an additional trick to be able to implement the collision detection for plankton eating using the physics engine: By default, physics are executed on the server and authorative there. However, as the plankton particles do not even exist there but are on the clients only, we added a local invisible mouth entity to the otherwise networked fish. This way client side physics works for detecting collisions of the fish mouths and the plankton.

Creating this setup obviously required designing and implementing the code with networking in mind -- in this case, the system definitely does not hide all the intricacies of networked games from the developer. The same uniform programming model is applied, certain entities are just configured to the local-only mode. Also the fact that in the Tundra SDK we have the same API both in the server and client executables (the core is the same) enabled an incremental development path here: first all the functionality was server side, but as the amount of networking grew to be too much, it was quite straightforward to change the same code to be executed on the client side only instead. As possible improvements for the future, both automated interest management to optimize network messaging, and easy robust ways to configure replicated vs. local execution are interesting.

(analysis of the Ludocraft's Circus code?)

The implementation in Tundra SDK

The Tundra SDK is a complete platform for networked 3d applications. It is built entirely using the inherently networked entity model described in this article. Here we give a brief overview of Tundra overall, and describe how the entity system works for application developers there.

Tundra core is written in C++ using several open source libraries: Ogre3d for the 3d scene and rendering, Qt for cross-platform support, GUI, event system and scripting support, Bullet for physics, OpenAL for audio, kNet for networking etc. It is a modular system where almost all the basic features are in optional plugins, and developers can write their own either for some new generic functionality or their own proprietary game logic and functionality. It supports scripting with Javascript and has optional support for Python modules as well. The same codebase is used both for servers and clients, and can be used standalone as well for single user applications.

The visible 3d scene and the custom application logics are typically made within the entity-component system, but other areas of functionality such as handling user input devices in GUI clients, manual asset downloads or dealing with network connections in the server are exposed as a set of core APIs.

To add a piece of functionality to a scene, a developer typically introduces a new entity-component type, in a plugin which also contains the code for handling that component. This is also how we have integrated several open source libraries: physical objects simulated by Bullet have a RigidBody component, the SkyX sky and clouds visualization Ogre plugin introduces a SkyX component with data such as the current time and the hour of sunrise as data attributes. Besides the automatic network replication of the attributes, Tundra core also can save and load the entity data to files (binary or xml), and provides a powerful basic GUI tool for working with components with an automatically generated interface (XXX add figure of entity-component editor, perhaps mention multi-editing).

This all works quite beatifully on the C++ level, but typically custom application functionality is implement in Javascript where the extensibility with custom components is not so well exposed. Currently no new component types can be added in dynamic code, but they all have to be defined in C++ at compile time. There is a special component called DynamicComponent to deal with this issue, and it basically allows human GUI users or Javascript code to define new components at runtime, but the API for defining new component types that way is awkward and there is no way to register new types with this mechanism so that they would work identically to the C++ written components in the GUI editor.

In fact, the original implementation of example 1. Pong game did not use the attribute system at all for game state, but instead was creating a Javascript dictionary called GameState and serializing that with JSON to send over the net with an entity-action. This came as a bit of a surprise for the author of this article when reading the code, written by another developer. When interviewed for this study, the developer revealed that it was due to the poor support for defining new components with the Javascript API in Tundra. If he had written the game using C++, he probably would have followed the pattern described in the example here and utilized the attribute system. In the version in the example here we work around the problem by declaring the PongGameState component in the application XML, in which case it exists already when the Javascript code is executed so it does not have to define it. But often it is better to have the component definitions in the program code, so definitely making good scripting support for that in Tundra is needed. Besides the definition of components (the attributes and their types), also hooking handlers directly to changes in certain attributes is lacking in the API -- this problem is visible also in the version here. (XXX We sketch and plan an API for that in the future work section of this article?)

There are various other stumbling blocks in game development with Tundra currently too, some of which are specific to networked environments. One is writing a script to some entity which further manipulates other entities in the scene. Especially if the script is to be executed on the client side, a naive implementation can fail to initialize when it is executed before the target entities had been replicated to that client (e.g. the scene.GetEntityByName("target") returns null unexpectedly and the rest of the code fails). In such cases we currently need to monitor the onEntityCreated signal to see when the entity of interest enters the scene. It may be possible to help there situations with better initialization orders and conditions, for example executing scripts only after a scene is completely instanciated, but that can be difficult in large worlds which are always only partially replicated to clients. One solution might be a more declarative programming approach, where relationships between entities and references to them are just declared and work, without manual procedural (/imperative?) code to get the references.

(.. other points and perhaps issues from tundra dev, what?)

(old, somehow nice partly:

Tundra applications are written against the Tundra Core API and utilizing the Entity-Component scene model. The platform takes cares of the networking basics, so that an application developer does not necessarily need to even know about connections, not to mention dealing with implementing own server and client applications somehow. When the application is run on a server, all clients due to the nature of the shared environment participate in the same session and see everything identically (and when they don't its' a bug and we must file an issue :p) <-- scrap that stupidity, it's just like scripting in any scriptable MMO .. or modding a FPS, using engine like Unreal or Quake. so can just put briefly and ref to something perhaps too, for clarity hopefully).)

The underlying networking for the entity system in Tundra SDK

The main focus in this article is to analyze the abstract entity-component model regarding the ease of development of networked multiplayer games. However the idea both with the theoretical model and the concrete implementations withing realXtend is to provide a system that really works in practice, is efficient and robust enough for commercial games and other applications. To this end, in this chapter the focus is on the concrete networking layer.

The replicated entity system with the generic attribute synchronization is implemented with a set of messages in Tundra, namely: CreateEntity, CreateComponents, CreateAttributes, EditAttributes, RemoveAttributes, RemoveComponents, RemoveEntity, CreateEntityReply, CreateComponentsReply and EntityAction (source: SyncManager:HandleKristalliMessage switch(messageId)).

Tundra uses the kNet library for transport, and kNet supports using both TCP and UDP. kNet allows sending arbitrary messages and features efficient serialization of basic data types. This way Tundra plugins can use own custom messages for efficient communications (currently this is limited to C++ plugins only).

Upon a new client login to a server, typically the whole scene state is replicated to the client using these messages. So the overall efficiency of entity-component-attribute creation is of importance. Then during the lifetime of a connection entities are typically not created nor removed that aggressively, but there can be constant streams of changes to attributes, so the efficiency of the EditAttributes message is crucial.

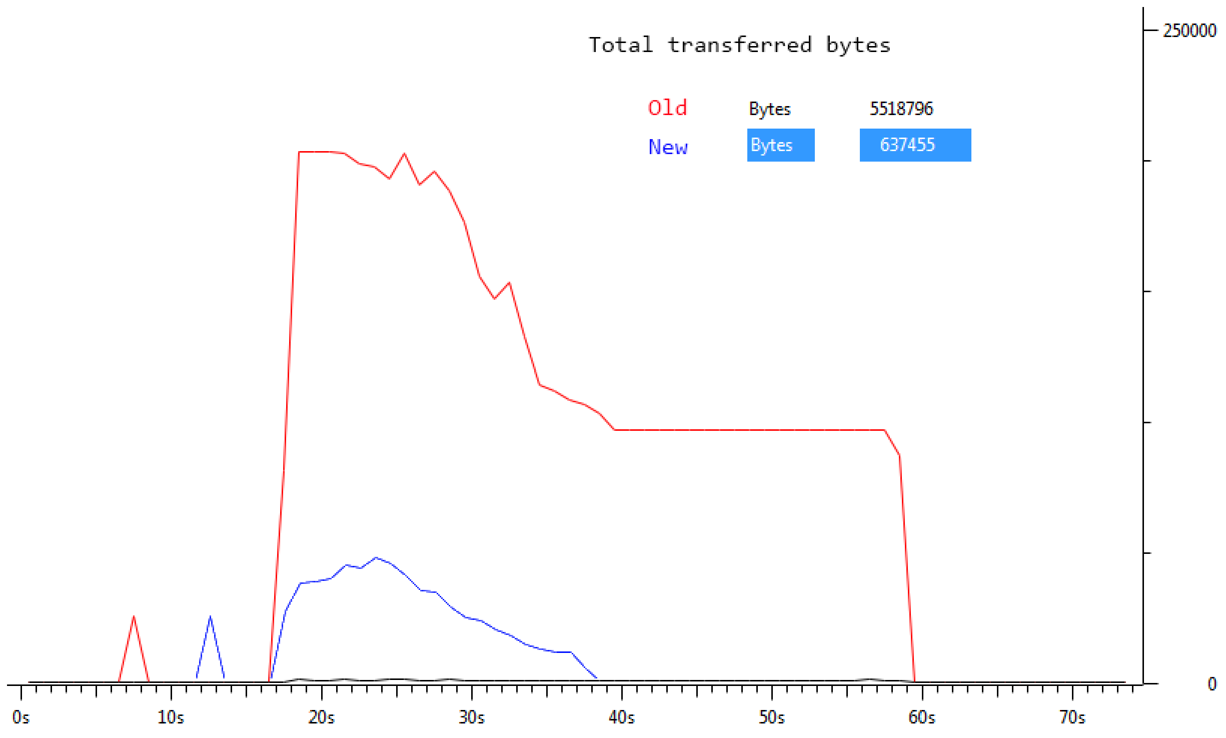
Besides the generic attribute sync, the Tundra SDK implements a custom message and corresponding logic for moving rigid bodies (RigidBodyUpdate). It utilizes motion interpolation and extrapolation (dead reckoning), and the physics module for non-authorative collision detection on the client side.

In fact in the first versions (1.0 - 2.3(?)) Tundra did not have a special message for moving objects, but the generic attribute synchronization was used for that as well (the floating point values in the transform attribute of the placeable component). The fact that we were able to have tens of simultanously moving objects with several client connections using that naive mechanism is some anecdotal evidence for the efficiency of the generic attribute synchronization. A generic optional attribute interpolation mechanism was made for smooth movements.

However, object movement was clearly such a common case and a bottleneck in many applications that the custom solution for it was required. The movement synchronization is essentially about synchronizing the linear and angular velocity vectors, only when they change, instead of trying to stream the resulting position all the time. Also specific custom messages have less overhead, as the message id already defines the target of the incoming data. With the generic attribute synchronization message, the message data has to identify the specific attribute that is being modified.

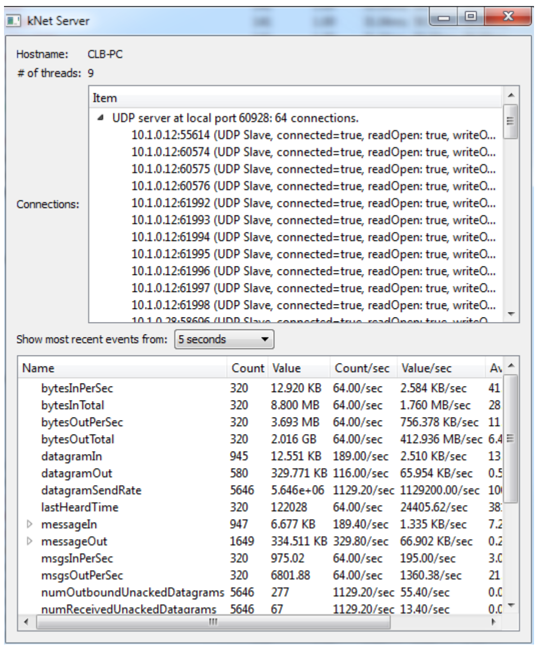
Network bandwidth measurements

We have conducted basic measurements of the network bandwidth usage of Tundra. With the original system of just using the generic attribute synchronization also for object movements, a single update was about 70bytes/update. The new rigid body streaming code averages at about 11bytes/update. This typically allows a far larger number of concurrent clients on a server.



Comparative profile of the old and new object movement code in the Tundra Physics demo scene

With the new system, user counts as large as 64 users are doable, but it largely depends on what is running in the scene.



Hostname: CLB-PC
# of threads: 9

Connections:

```
Item
◢ UDP server at local port 60928: 64 connections.
    10.1.0.12:55614 (UDP Slave, connected=true, readOpen: true, writeO...
    10.1.0.12:60574 (UDP Slave, connected=true, readOpen: true, writeO...
    10.1.0.12:60575 (UDP Slave, connected=true, readOpen: true, writeO...
    10.1.0.12:60576 (UDP Slave, connected=true, readOpen: true, writeO...
    10.1.0.12:61992 (UDP Slave, connected=true, readOpen: true, writeO...
    10.1.0.12:61993 (UDP Slave, connected=true, readOpen: true, writeO...
    10.1.0.12:61994 (UDP Slave, connected=true, readOpen: true, writeO...
    10.1.0.12:61995 (UDP Slave, connected=true, readOpen: true, writeO...
    10.1.0.12:61996 (UDP Slave, connected=true, readOpen: true, writeO...
    10.1.0.12:61997 (UDP Slave, connected=true, readOpen: true, writeO...
    10.1.0.12:61998 (UDP Slave, connected=true, readOpen: true, writeO...
```

Show most recent events from: 5 seconds

Name	Count	Value	Count/sec	Value/sec	Av
bytesInPerSec	320	12.920 KB	64.00/sec	2.584 KB/sec	41
bytesInTotal	320	8.800 MB	64.00/sec	1.760 MB/sec	28
bytesOutPerSec	320	3.693 MB	64.00/sec	756.378 KB/sec	11
bytesOutTotal	320	2.016 GB	64.00/sec	412.936 MB/sec	6.4
datagramIn	945	12.551 KB	189.00/sec	2.510 KB/sec	13
datagramOut	580	329.771 KB	116.00/sec	65.954 KB/sec	0.5
datagramSendRate	5646	5.646e+06	1129.20/sec	1129200.00/sec	10
lastHeardTime	320	122028	64.00/sec	24405.62/sec	38
▷ messageIn	947	6.677 KB	189.40/sec	1.335 KB/sec	7.2
▷ messageOut	1649	334.511 KB	329.80/sec	66.902 KB/sec	0.2
msgsInPerSec	320	975.02	64.00/sec	195.00/sec	3.0
msgsOutPerSec	320	6801.88	64.00/sec	1360.38/sec	21
numOutboundUnackedDatagrams	5646	277	1129.20/sec	55.40/sec	0.0
numReceivedUnackedDatagrams	5646	67	1129.20/sec	13.40/sec	0.0

A kNet server with 64 connections (XXX: Jukka - doing what?)

This was the first optimization, a basic sanification, made to allow for more moving objects and client connections to Tundra scenes. After that, the focus has been moved to apply the common basic techniques to deal with larger worlds and lots of traffic, namely scene partitioning and importance based interest management. That work is described later in this chapter.

Overall performance and scalability

about scalability & performance in general:

```
From:   Jukka Jylänki <jukka.jylanki@ludocraft.com>
[realXtend-dev] Scalability study for Tundra.
Date:   April 18, 2012 3:47:57 PM GMT+03:00
```

https://groups.google.com/forum/?fromgroups#!topic/realxtend-dev/Lzzx_hZu38I%5B1-25%5D

- performance

interest management

(worked on at Chiru -- report preliminarily here. also what Ali is doing at Ludocraft)

- not all entities everywhere always
- sync rate adapted based on importance -- crossref with the example 2 where had to deal with bandwidth manually (being able to do local entities for visual effects etc. is probably still good, even if perfect magical IM was there)

Evaluation and Discussion

(compare with sirikata / emerson and others in the related work. unity?)

Alternative implementations: the entities to web browsers and XMPP as well?

Besides the native C++ Tundra SDK, we have also implemented early versions of alternative clients on the web (html5) and Flash/AIR runtimes. For continuing this work it is imperative to know whether and how the entity model reaches the goal of making multiplayer game development productive. (XXX: isn't it also vice versa -- it can actually be used in the evaluation here to judge the model? restructure to earlier in the article? XXX) That is: do we find this entity system so good that we want to have the same as the basis also for browser based games?

In the web browser technology based client, so called WebNaali, websockets are used for the communication. We implemented a server side plugin for that (originally in Python) and a corresponding client stack in Javascript, using a WebGL engine called GLGE for the 3d rendering. We begun by synchronizing the whole scene state at login to a Tundra server using the EC model (at first with Tundra XML). Then we implemented a single message for generic attribute synchronization, and enough handling code for Placeable updates to work (note: this was originally made before the special RigidBodyUpdate system existed). Also, we implemented the single message required for entity-actions. That was required for the avatar and chat functionality which was the customer requirement for the first WebNaali test service. The Tundra avatar and chat applications work by client sending commands to the server, which then in the case of the avatar moves the character (resulting in placeable sync back to the client), and in the case of chat the server also uses entity-actions to send the chat messages to clients. This was quite simple and quick to do, the whole WebNaali 0.1 code is only N lines. A video demonstrating simultaneous views to the same scene with a WebNaali and a native Tundra client, demonstrating those avatar and chat functionalities (without animation state sync in WebNaali) is at:

. The entity-component model has been straightforward to implement, thanks to the genericity of attributes and actions, we have been able to make ground in WebNaali for a wide range of applications by implementing only a few network messages.

On the Flash front, there is currently a different platform candidate under the realXtend open source umbrella. In so-called Lehto, instead of Tundra we are utilizing plain XMPP as the server and networking backend. In the networking level, Lehto clients are just normal XMPP clients, utilising the XMPP

Multi-User Chat (MUC) extension for group sessions. This was practical in a customer project where only chat communication and rare presence updates sufficed, and no physics nor scripted application logic were required on the server side, so Tundra was not needed. We have not yet implemented any entity system in Lehto, currently it is very simple and can just load a static scene from a normal static geometry file, only features the hardcoded chat (standard xmpp) and simple avatar position updates using a simple message via a hidden non-human control MUC. XMPP is very verbose, so we limited avatar sync rate to 1 second, which sufficed for the application (a virtual gallery system for Berlin Gallery Weekend 2012, see: XXX). However with the XMPP Stream Compression extension the bandwidth may reduce dramatically -- possibly enough for realtime gaming? We have not tested the stream compression yet, as it was not required for the gallery application and the Flash client library used, XIFF, did not support it at the time. However XIFF got streaming compression support now (August 2012), and there are mature implementations in other languages (at least Java) so it could be tested. Lehto development continues during autumn 2012 and we will certainly consider implementing the entity system and generic attribute synchronization if they are required for some applications.

Notes / References

(NOTE: below is selected copy-pastes from potential references, mostly not original text!)

Greger Wikstrand, Lennart Schedin and Fredrik Elg [9] gave three hypotheses before they did their Pong game experiment in a simulated mobile phone: ”Delay effort”, ”De- lay action” and ”Delay performance”. The experiment put eyes on significant effects on four independent variables: enjoyment, mental effort, net distance and paddle move- ---

---

Avango is a framework for building distributed virtual reality applications. It provides a field/fieldcontainer based application layer similar to VRML. Within this layer a scene graph, based on OpenGL Performer, input sensors, and output actuators are implemented as runtime loadable modules (or plugins). A network layer provides automatic replication/distribution of the application graph using a reliable multi-cast system. Applications in Avango are written in Scheme and run in the scripting layer. The scripting layer provides complete access to fieldcontainers and their fields; this way distributed collaborative scenarios as well as render-distributed applications (or even both at the same time) are supported. Avango was originally developed at the VR group at GMD, now Virtual Environments Group at Fraunhofer IAIS and was open-sourced in 2004. An in-depth description can be found in here.

- a publication: Improving the AVANGO VR/AR Framework — Lessons Learned Download, presented at the 5. GI VR/AR workshop. The slides Download are also available.

http://www.avango.org/raw-attachment/wiki/Res/Improving_the_AVANGO_VR-AR_Framework--Lessons_Learned.pdf

- http://www.avango.org/wiki/Concepts

NOTE: Avango concepts seem quite similar to tundra - 'fields' is a bit like our attrs, are autoserialized etc., and there are connections which are perhaps similar to qt signal conns .. the example there is a proximity sensor

---

Pong with a multiplayer Flash platform: multiplayer pong example & tutorial http://www.unionplatform.com/?page_id=1229

"Union Pong consists of a server-side room module written in Java, and a Flash client-side application written in pure ActionScript with Union's Reactor framework. The room module is responsible for controlling the game's flow, scoring, and physics simulation." jne - client attribuutteja näemmä settailee - näemmä aika paljon pitää tuolla ite hanskailla attribuuttien muutoksien lähettelyä ja vastaanottoa

---

homura (appears dead since 2010, was started in 2007 -- bbc and uk edu, a bit similar to realXtend, with games focus) from: http://java.cms.livjm.ac.uk/homura/links.php Dennett C., El Rhalibi A., Merabti M., Price M.,"Koku: State Synchronisation System for Networked Multiplayer Games", 6th International Conference in Computer Game Design and Technology (GDTW), Holiday Inn, Liverpool, UK, 12th - 13th November 2008. http://java.cms.livjm.ac.uk/homura/dist/docs/Paper-GDTW2008-Koku.pdf

- NOTE: this was apparently mostly interest management like, about dividing a large world hierarchically and about granularity of required information etc. -- could be mentioned in the LVM fishgame bandwidth optimization treatment!

from: http://www.cms.livjm.ac.uk/pgnet2010/MakeCD/index.htm NHUGS: Towards scalability testing for MMOGs within an extensible, open architecture Carter, C., El Rhalibi, A., Taleb-Bendiab, A., Merabti, M., Liverpool John Moores University http://www.cms.livjm.ac.uk/pgnet2010/MakeCD/Papers/2010022.pdf

homura middleware, from http://java.cms.livjm.ac.uk/homura/dist/docs/Paper-GDTW2008-NetHomura.pdf "The Development of a Networking Middleware and Online-Deployment Mechanism for Java based games."

"The NetHomura middleware integrates with the Homura Engine to create a GameStateManager to control the game. This manages an internal stack of HomuraGameState instances. HomuraGameState is an abstract class which implements the game loop of each state, providing methods for initialisation of content, handling user input and updating the state of the game world (members of the scenegraph), and a rendering the scenegraph to screen. The NetHomura games are comprised of concrete implementations of this class (e.g. MainMenuState, LoadingState, PuzzleGameState, etc.). The middleware provides an additional implementation, NetState, which encompasses the additional interactions of a network game, by adding methods to receive messages, send messages, join and leave games. The NetState class uses an instance of the middleware's NetManager class, which handles peer-management facilities such as discovering available game sessions, creation of new game session, tracking and modifying persistent, shared data objects used within the game, managing references to connected peers, sending messages to particular peers and retrieving messages that are received from peers. The NetManager also handles session control, such as disconnecting and joining into both the entire network and game sessions. The role of the game developer using the middleware is to create game-specific messages which inherit from the base NetHomuraMessage class. This class encapsulates the in-game messages sent between peers, and using the NetTools class to construct efficient managements using the functions to efficiently serialise Java object into messages. These messages can then be broadcast using the NetManager. The middleware also provides the concept of GameSessionAdvertisments, which are used to create and communicate the details of a particular game session to other peers so that they can participate in a session. "

---

general, should get

T. Hsiao and S. Yuan, "Practical Middleware for Massively Multiplayer Online Games," IEEE Internet Computing, vol. 9, 2005, pp. 47-54.

explore.ieee.org/xpl/login.jsp?tp=&arnumber=1510604&url=http%3A%2F%2Fieeexplore.ieee.org%2Fxpls%2Fabs_all.jsp%3Farnumber

J.D. Pellegrino and C. Dovrolis, "Bandwidth requirement and state consistency in three multiplayer game architectures," Proceedings of the 2nd workshop on Network and system support for games, Redwood City, California: ACM, 2003, pp. 52-59;

http://portal.acm.org/citation.cfm?id=963900.963905&type=series.