Tackling online game development problems with a novel network scripting language

George Russell
Codeplay Software Ltd.
45 York Place
Edinburgh, Scotland
george@codeplay.com

Alastair F. Donaldson
Codeplay Software Ltd.
45 York Place
Edinburgh, Scotland
ally@codeplay.com

Paul Sheppard
ITI Techmedia
Glasgow, Scotland
paul.sheppard@
ititechmedia.com

ABSTRACT

We describe a novel scripting language for writing bandwidth-efficient online game logic. The language facilitates the development of deterministic, concurrent, distributed games, with assurances of consistency maintenance between clients and server. Our approach allows for increased simulation accuracy when compared to dead reckoning, and removes the need to write code to repair distributed state inconsistencies, or to explicitly transfer data over a network.

Categories and Subject Descriptors

C.2.4 [**Computer Communication Networks**]: Distributed Systems; D.1.3 [**Programming Techniques**]: Distributed systems; D.3.3 [**Language Classifications**]: Concurrent, distributed, and parallel languages

General Terms

Experimentation, Performance, Algorithms, Measurement

Keywords

Computer games, Concurrency, Distributed games

1. INTRODUCTION

An online game is a concurrent, distributed application. As a result, programmers of online games face new complexities related to network latency, out-of-order delivery, and partial failure. Relatively few programmers are experts in distributed and concurrent programming, which introduce new classes of nondeterministic bugs such as deadlocks and race conditions, that are hard to reproduce in a test or development environment. Problems of nondeterminism are complicated in heterogeneous networks, where identical program logic executed on different architectures may produce subtly different results.

These issues concern correctness. For a game to be responsive, careful consideration must be given to issues of bandwidth and network latency. The state of an online game is typically large, while the bandwidth available between client and server is limited. This restricts the frequency with which state updates can be issued to clients, and renders it impractical to send updates for the entire game state [4]. Although interest management [3], delta encoding [9] or dead reckoning [1, 10] can often be used to achieve an acceptable frame rate, for game scenarios with a large population of dynamic objects on a client, the quantity of data to be transmitted can be excessive. According to [14], "If you have more updating data than you can move on the network, the only real option is to generate the data on each client." Network latency can lead to significant delays between a user initiating an action and perceiving the result of that action. This may render a game frustrating or unplayable. Techniques for masking latency work by allowing clients to simulate game entities up to a certain level of inaccuracy, computing future attributes of game entities via prediction or extrapolation based on recent history. Due to limitations in prediction algorithms and the unpredictability of user input, latency masking techniques result in inconsistent views of the game world between clients. Brief periods of inconsistency can be allowed, but must be recognised and corrected, e.g. by a *timewarp* algorithm [11].

As well as being correct and efficient, an online game should scale to support a large population of game objects and players.

In this paper we present the *network scripting language* (NSL), a scalable, bandwith-efficient programming language for online game logic which hides issues of concurrency, distribution and latency from the programmer. We provide an overview of the language, a theoretical discussion of the reduced bandwidth offered by NSL, and promising experimental results for a prototype.

2. THE NETWORK SCRIPTING LANGUAGE

We have identified two major problems in the implementation of online games: the bandwidth requirements for transmission of state updates, and inaccuracies resulting from the use of dead reckoning and latency compensation techniques. Our solution is as follows: the initial state of a given object is issued to all interested clients. In addition, the clients are issued with deterministic code to simulate the object over time. Replicating code with objects allows clients to independently update their area of interest consistently with updates computed by the server.

Our approach promises a potential reduction in bandwidth requirements: instead of transmitting a sequence of state updates from the server to a client, data transfer between server and clients is now only required to communicate user actions, remote inputs, and to handle objects moving in or out of a client's area of interest. Replicating the computation of state updates provides accuracy and consistency, as state updates for game entities are computed consistently, not extrapolated and subsequently corrected. Our approach to masking latency in the propagation of user inputs between clients is similar to *timewarp* [11]: computation proceeds optimistically

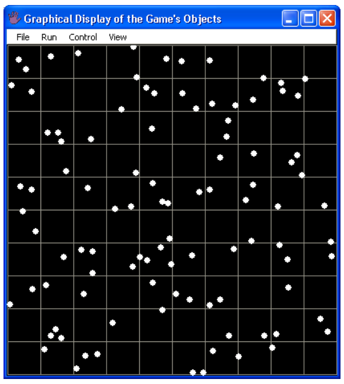

Figure 1: An aerial view of PointWorld.

on the assumption of no input, and the state of client-side entities is corrected by re-execution as inputs are received.

To achieve the desired features of determinism, replication, and self-correcting timewarp execution of game logic, together with ease of development we have designed and implemented a novel domain-specific language. The network scripting language (NSL) is an object-oriented language with support for message passing, deterministic concurrency and distribution, to be embedded within game clients and to act as a server for game logic. A language can provide and enforce guarantees that would be difficult to obtain and maintain with existing programming languages and middleware.

We now introduce NSL and discuss the novelties of the language which lead to the desirable properties discussed above.

2.1 An Overview of NSL

NSL programs are written in a Java-like syntax, which we illustrate throughout the paper via a simple distributed world called PointWorld. The screenshot of Figure 1 shows this world viewed from above. A population of game entities, or *points*, shown as white circles, are initialised with a position and velocity. Points in the world interact via collision response: upon colliding, points recoil from one another with some degree of randomness in the angle of departure. The world is divided into *cells*, indicated by the grey lines in Figure 1.

An NSL object is an *active object* [13], a lightweight process with its own independent thread of control. The PointWorld example contains an object per *point* and per *cell*, with an additional object to bootstrap and manage the world and its population.

Figure 2 shows how the `Point` class is defined in NSL. Control flow for an object starts in the constructor, allowing the object to perform initialisation, optionally with parameters from its creating environment. Control then passes to the object's `execute` method (Line 15 of Figure 2). This method conventionally iterates to compute a state update for a single frame. The actions performed by a *point* per frame are as follows: the *point* updates its position to move within the world, then checks for collision with other points in the same and neighbouring cells. Movement may cause the *point* to leave one cell and enter another, in which case the *point* notifies the associated cells. Finally, the *point* either terminates (in response to a message from another object), or suspends execution until the next frame.

The receive statement at Line 21 in Figure 2 shows a test for receipt. Messages are matched by type, e.g. `Exit`, and appear as a local variable in the receive block scope (`e` in our example). If the *point* receives an `Exit` message it terminates via `quit`. If there is no message, execution jumps to the end of the receive block.

```
 1 class Point {
     public var vec2 pos, vec2 vel;
 3   public var Cell cell;

 5   Point constructor(Cell container
                      , vec2 position
 7                    , vec2 velocity) {
         pos  = position;
 9       cell = container;
         vel  = velocity;
11       /* Points are inside a Cell */
         send(cell, NewContent(previous));
13   }
     /* Methods */
15   void execute() {
       var float dt = 1.0/FPS;
17     loop {
           move(dt);
19         checkCollision(dt);
           checkForChangeOfCell();
21         receive Exit as e {
               quit;
23         }
           yield;
25     }
     }
27
     void move(float dt) {
29     pos += vel*dt;
       pos = checkAgainstBounds(pos);
31   }

33   void checkCollision(float dt) {
       /* Checks only within 1 cell for brevity */
35     const int N = len(cell.contents);
       for var int j = 0; j < N; j++ {
37       var Point2D p = cell.contents[j];
         if p == previous {
39         /* No self collision! */
           continue;
41       }
         if collidesWith(p, dt) {
43         /* Respond to one collision by moving */
           var float angle
45           = -1.0 * angleTo(pos, p.pos);
           angle += (rand(HALF_PI*100.)/100.)
47              - HALF_PI/2.;
           var float velLen = len(vel);
49         vel.x = sin(angle)*velLen;
           vel.y = cos(angle)*velLen;
51         move(dt);
           return;
53       }
       }
55   }

57   void checkForChangeOfCell() {
       if not cell.isInCellBoundary(pos) {
59       var Cell old = cell;
         cell = cell.findCellLocation(pos);
61       send(old , RemoveContent(previous));
         send(cell, NewContent   (previous));
63     }
     }
65 }
```

Figure 2: Class declaration for a *point*.

The `yield` statement at line 24 asserts that execution should be suspended for this frame, and resumed for the next frame after the `yield` statement.

The `move` method for a *point* computes a new position from the current position, time step and velocity, as shown on line 28 of Figure 2. After moving, a *point* checks for and responds to collisions with other points in its cell via the `checkCollision` method. This requires access to the state of the points and the containing cell. The NSL semantics (described in Section 2.2) ensure that this access is safe in a concurrent environment. To move from cell to cell within the game world, a point must communicate with both cells concerned by message passing, via the `send` operation.

Our example illustrates NSL code for an actor within a simple game world. In summary, an NSL actor may:

- Read and mutate local state, e.g. to update position
- Suspend execution for $k > 0$ frames, or until an input message is received
- Send a message to another object, e.g. a command or request
- Read another object's state from the previous frame
- Invoke a non mutating method upon another object
- Test for receipt of a message or messages
- Terminate
- Spawn a new parameterised object.

An object terminates by a `quit` command or by returning from the `execute` method. Abnormal termination via `assert` or unhandled exceptions is also possible.

2.2 Language Semantics

In this section we summarise important aspects of the NSL semantics which guarantee deterministic execution in a concurrent, distributed environment.

Guaranteeing determinism in the presence of concurrency is a complex problem. Our solution is to *delay* the visibility of state updates. Given distinct objects o_1 and o_2, if o_1 reads a public field x of o_2 during execution of frame n then o_1 receives the value for x in frame $n - 1$. The language guarantees that values of variables in frame $n - 1$ are not modified once execution of frame $n - 1$ has completed, thus o_1 can read the variable x at any point during frame n and always get the same result.

Without this restriction, concurrently executing objects would be able to read the changing state of other objects, leading to nondeterministic behaviour dependent on the exact timing of reads. The programmer would be required to write complex, error-prone locking code to synchronise object actions. Restricting access to non-local data to refer to the preceding frame avoids these problems: this data is immutable, and can thus safely be accessed concurrently without locks. This eases programming of game logic, and allows objects within a frame to be safely and deterministically executed in any order. Furthermore, execution of multiple objects can be easily distributed over multiple cores.

As discussed in Section 2.1, communication between objects is achieved by message passing rather than by direct manipulation of state. Messages are strongly typed records which may contain references to other objects. The ability to pass object references via messages leads to a dynamic interconnection topology: an object can hold references to other objects, these references can be passed around, and two objects are able to interact if one holds a reference to the other.

Each object has a mailbox per frame for incoming messages sent during *previous* frames. This allows determinism under concurrency: if point p leaves cell c_1 for cell c_2 in frame n then appropriate messages are sent to c_1 and c_2 during execution of frame n. However, these messages are not received and processed by the respective cells until frame $n+1$. This avoids nondeterminism which could arise due to precise message delivery times if messages were sent and received within the space of a single frame. The order of messages is also deterministic, and is not necessarily the order in which messages are sent: multiple messages from a single sender are received in order; messages from two or more senders are received in a fixed, relative order.

Delaying the visibility of state changes means that an unmodified copy of the game state for frame n must be available during execution of frame $n + 1$. Therefore, NSL object state is copied between frames for non-terminated objects. It is possible for a reference to an object to persist after the object referred to has terminated. If object o terminates in frame n then the state of o can still be accessed via references in frame $n + 1$, but in subsequent frames an access to o will raise an exception.

We require serial program determinism as a basis for concurrent determinism. In particular, we require that floating point arithmetic and integer overflow semantics do not vary between NSL implementations on distinct platforms.

An NSL program is executed by the NSL runtime system. An object constructor invoked in the global scope of the program creates an object to bootstrap the initial frame of objects. The game state for a frame is comprised of the states for all objects active in the frame, and the state of an object includes its last point of execution. Running the game logic for a frame involves executing code from the point of suspension in the previous frame until the object suspends itself, or terminates. The result of this execution forms the game state for the next frame.

The semantic restrictions required by NSL can be summarised as follows:

- Global shared state is never mutated
- Reads from non-local state refer to data in the preceding frame, which are immutable in the current frame
- Non-local method invocations operate on data in the preceding frame, and may not mutate this data
- Messages sent in a given frame are received in the subsequent frame
- Mailbox messages for an object are ordered in a consistent manner
- Mutating an NSL object state in execution of frame n must not mutate the state of the object in any frame $m \neq n$
- An NSL implementation must ensure the consistency of basic language operations across all supported platforms
- NSL calls into native libraries must observe the restrictions required for determinism.

Adhering to these restrictions makes concurrent, distributed software easier to write, debug, maintain and reason about, since many large classes of potential bugs are immediately precluded from manifesting during program execution. The restrictions on mutation of non-local data ensures independence of computation for distinct objects in a single frame, thus execution of an NSL frame is readily broken down into small, independent pieces. This allows relevant subsets of game logic execution to execute on clients, interacting with logic on the server only in specific, well-defined manners.

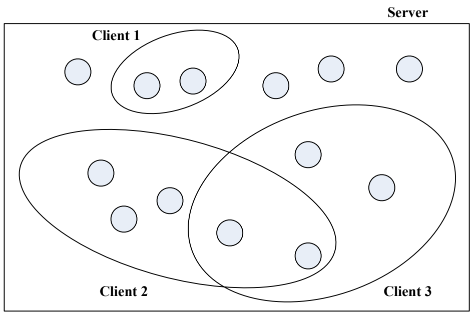

Figure 3: Replicating subsets of server computation on clients.

3. DISTRIBUTED NSL

A naïve distributed NSL implementation would replicate the entire game state on to each client and execute the clients in lock step with the server, ensuring that user inputs between frames n and $n + 1$ reach all participants before execution of frame $n + 1$ proceeds. A client could then track the evolving state of every game object by resuming execution from its suspension point for frame $n + 1$. Such a naïve approach is simple, but flawed. While it prevents inconsistent views of game state, it decouples synchronisation from wall clock time, and couples simulation frame rate with network delay. This would reduce the speed to that of the slowest player. Also, the entire game state could be too large or computationally intensive for replication on a typical client, and forwarding all client inputs to all other clients would scale poorly for large client populations.

Instead, our NSL implementation uses a more sophisticated approach based on a combination of optimistic synchronisation and interest management techniques, which we now describe.

3.1 Interest Management of Computation

Interest management logic in a script permits the server to dynamically select, for each client, a subset of objects to be replicated to each client. This subset should represent the *area of interest* of a client – the view of the game world for a player. Figure 3 shows a Venn diagram illustrating an NSL program of 13 objects. All objects execute on the server, indicated by the enclosing rectangle. Three subsets are identified, representing subsets of the population replicated on clients. An object in a given subset executes both on the server and the associated client. Subsets may overlap, in which case the object executes on the server and multiple clients. Some objects do not belong to any client subset – these objects are tracked by the server only.

Subsets replicated on a given client are dynamic. An object can enter the subset for one of three reasons: the server determines that the object is now relevant to the client (e.g. when the object enters the area of interest); an object currently in the subset requires the new object to perform a computation, or the object is created by an object in the subset. An object leaves the subset when the server determines it should do so. The programmer implements a distinguished server-side object to deal with player connections and handle game-specific area of interest management.

Interactions between local and remote objects are treated identically at the program level, and consist either of message transmission, non-mutating field accesses or method calls. If object o_1 sends a message to object o_2 then the server sends this message transparently across the network to any clients which replicate o_2 but *not* o_1. If a client replicates both o_1 and o_2 then the transmission of the message between objects can be handled locally. Note that the server tracks *all* game objects, so it is not necessary for clients to forward messages between objects to the server. A field access or method call requires the accessed object to be fetched if not already present locally.

Messages representing user input from the players of the online game are created outside the NSL system. A user input message is injected into NSL on the associated client, addressed to an NSL object (e.g. the player's avatar) in a specific frame. These messages are forwarded on to the server. In order to maintain determinism, the server forwards the messages to any other clients which replicate the associated object. The forwarding of input messages only to interested clients limits the quantity of input events retransmitted from the server, thus reducing network bandwidth.

3.2 Optimistic Synchronisation and NSL

Network latency may result in the late delivery of input messages. Optimistic synchronisation [6, 11] is used to correct this, to ensure consistency of state across clients and server. An important novelty of our approach is to embed support for optimistic synchronisation into the language itself; the frame based semantics permit self correcting speculative execution of client-side NSL.

Each frame is repeatedly executed as new network inputs for that frame are received, starting from the frame with the oldest unapplied input, proceeding up to the most recent frame, until the client's state for a frame is consistent with the server, and therefore correct. Unlike timewarp, we aggregate roll backs as a result of messages being addressed to frames, and can repair multiple inconsistencies in performing a single roll back. Further, NSL objects may update their state without external events, and so our notion of virtual time and order of events is based on frame order and message ordering within frames, in contrast to the event driven execution of timewarp.

Notably, the script does not need to contain explicit code to handle roll back, reversal of actions or correction of state. The semantics of NSL allow the runtime system to perform the necessary actions to achieve consistency and correctness, whilst hiding the complexity of these actions from the programmer.

The client informs the server when execution of a frame and input sequence is complete. The server maintains a history of k frames for some $k > 0$ (specified by the server administrator), while the client maintains a history of frames from the most recently confirmed consistent to the most recently executed frame on the client. The server disconnects clients which fall behind by more than k frames. Given a player input, the server confirms to which frame this input should be applied. Clients execute speculatively, applying player inputs without waiting for the server, and re-executing these frames if speculation turns out to be incorrect due to latency in message transfer. Eagerly applying input can result in re-execution of many frames when latency is high; a local lag on application of player inputs reduces this problem, trading accuracy of client execution and reduced processing against responsiveness to user input.

4. EXPERIMENTAL EVALUATION

We have written an interpreter for NSL, to be used by C++ applications for loading scripts. An application using an NSL script may act as a server, client, or standalone component, and is able to control, interact with, and query objects. Our prototype uses TCP/IP for reliable network transport. A graphical replay debugger acts as an embedding application. We now present experimental results for bandwidth, server traffic, and multi-core acceleration for an example game world.

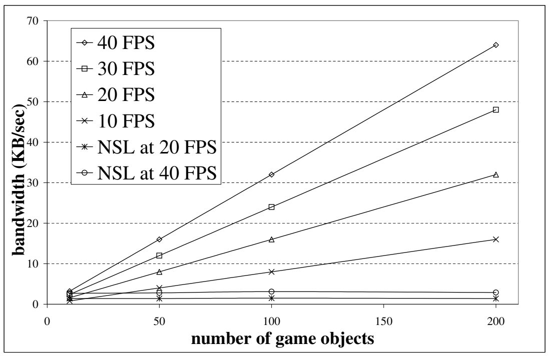

Figure 4: Comparing bandwidth requirements for theoretical state transfer and NSL for object replication.

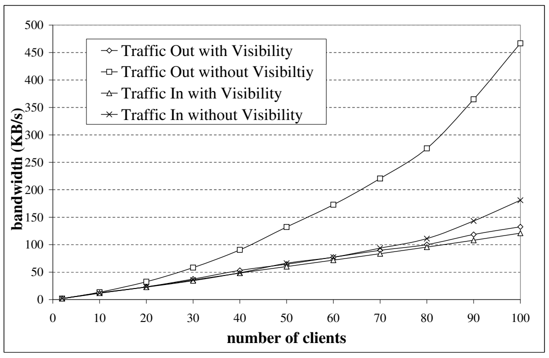

Figure 5: Server network traffic for frequent user input, with and without visibility management.

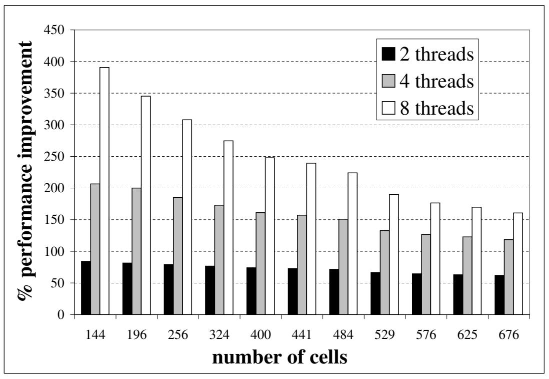

Figure 6: Performance scaling for multi-threaded byte code interpretation on an 8-core machine.

4.1 Methodology

In order to measure bandwidth requirements, we have written a set of distributed programs in NSL, based on a 2D game world containing moving and interacting players and NPCs. A 2D world is both simple to use for our experiments, and a reasonable simplification of many online games. We divide the world into equal-sized grid cells, to allow for scalable interactions.

We measure the bandwidth usage of the NSL programs and compare against the theoretical network bandwidth required for an implementation based on state transfers. Bandwidth measurements are obtained using a 3rd party bandwidth monitoring tool [12] which records, for a given machine, the maximum, and average incoming and outgoing bandwidth usage over a fixed time period. Average bandwidth is measured after initial connection handling traffic has passed out of the history range, and with unrelated network traffic eliminated where possible. We report data for a network of x86 PCs, running Windows XP. The server has a dual core 2.8 GHz Pentium D processor, and 1.5 GB RAM. Network clients are "bot" programs connecting via a LAN and sending sequences of input events read from files.

4.2 Results and Comparison

The screenshot of Figure 1 shows a simple NSL game world, divided into 20x20 cells, wrapping at the edges. 100 game actors move within this world, colliding with one another as they do so.

Figure 4 shows the bandwidth required to replicate a server game world on to a single client, with a varying world population and frame rate. The outgoing bandwidth required by the server for a state transferring implementation of our scenarios can be determined in bytes/sec using the following formula [4]: $n * s * f * c$, where n is the number of objects in play, s the average size (in bytes) of game objects, f the networked simulation frame rate for the game (in FPS), and c the number of clients. For our single-client scenario, this becomes $n * 8 * f * 1$, since 8 bytes are required to represent the coordinate pair required to render a *point*. For NSL, in the absence of client user inputs, network traffic consists of the client and server exchanging notifications of successfully executed game frames, and confirmation of absence of inputs for frames. Consequently, the per-client bandwidth is not proportional to the number of game world entities replicated, but rather to the FPS of the game. In contrast, a state transfer based system transmits a quantity of data proportional to the number of game objects and the frame rate of the game, leading to higher bandwidth requirements.

Figure 5 shows input and output bandwidth for an NSL server with between 2 and 100 connected clients, each generating an input event every second. Results are shown with and without restrictions on the server's forwarding of user inputs. When visibility management techniques are not employed, server input scales linearly with the number of clients, but outgoing traffic increases super-linearly. Scalability here is unproblematic for relatively small numbers of clients, but bandwidth requirements would clearly become prohibitive for many clients. Figure 5 shows that when interest management is enabled and clients are not visible to one another, outgoing server traffic grows linearly with increased client connections while input remains linear. With interest management, the server will relay inputs from client c only to clients for whom c is visible. The rate of outgoing traffic is still higher than for incoming traffic, since objects are pushed to clients as players migrate across the game world and the visible area changes. Provided that player objects do not congregate into a single area of interest, the scalability with respect to input events is improved.

We have also investigated the impact of concurrency on performance for a PointWorld program with 676 actors and varying numbers of cells, executed for 250 frames in isolation without an FPS cap, and using 1, 2, 4 or 8 byte code interpreter threads on a system with two 1.6GHz quad-core Intel Xeon E5310 processors and 2 GB RAM. Figure 6 shows the performance of our prototype interpreter in this multi-core environment. Performance scales as more cores are added, but not linearly as we would ideally like. While objects execute in parallel, runtime inter-frame actions are serial, and therefore not all computation exploits concurrency.

Scalability with the number of cores is best when objects perform significant work per frame. In PointWorld this corresponds to a scenario with a high population density, and a consequently high rate of interactions. Without interactions, the points perform relatively little computation to benefit from concurrent execution. Note that concurrent execution of NSL code is automatic, only requiring the user to set an environment variable specifying the maximum number of byte code interpreter threads to be spawned.

5. RELATED WORK

Dead reckoning [1, 10] or delta encoding [9] are well known means of improving bandwidth efficiency of state transfers. Existing interest management techniques [5] could manage the set of computations replicated on clients. The *timewarp* algorithm [11], a variant of which is incorporated in NSL, for correction of inconsistent client state is a known means of resolving inconsistent client views. Trailing state synchronisation [6] is another. .

Existing concurrent and distributed languages such as Erlang [2] and Stackless Python [15] lack the determinism of the NSL, but give credence to languages for highly concurrent distributed programs.

6. CONCLUSIONS & FUTURE RESEARCH

We have presented NSL, a novel scripting language which aims to ease programming of online game logic. To the best of our knowledge, ours is the first approach which works by replicating deterministic computations from the server to clients within a semantics enforcing consistency maintenance. Promising initial experimental results show that NSL can result in efficient use of network bandwidth, and can effectively exploit the power of multicore processors for frame execution.

Under certain conditions, an NSL program may not scale well to a large number of clients. Potential causes of poor scaling include: passing large data volumes via messages from server-only objects to objects replicated on clients; repeatedly culling and repopulating a client object set (e.g. due poor interest management), and replicating large portions of the game world on clients due to complex dependency chains through the object population. In addition, transmission of a resumable computation has a size overhead relative to pure data for a one-off transfer: if an object's tenure on a client is too brief, the object will not earn back the cost of its initial transmission.

The frame-based semantics permit potential optimisations for sharing of state between copies of frames maintained by a client, via a "copy on write" approach. This would reduce the amount of data which must be copied for each frame, reducing per-frame storage and processing requirements for large, fast-paced games. Existing work addressing issues of fairness and cheating in distributed games such as cheat proof event ordering could be integrated with NSL [7, 8].

This research is a step towards easing development of online games by abstracting over challenging programming areas of concurrency and distribution for game object simulation. Additionally, lowering bandwidth requirements can potentially increase the number of concurrent clients which can be supported in a game, or reduce the operating costs related to bandwidth usage.

The NSL implementation used for experiments in this paper is a prototype. We are keen to work with commercial partners on developing a highly performant, robust implementation, and using this implementation to develop an industrial scale online game.

Acknowledgments

This research was funded by ITI Techmedia as part of their online games development programme, which seeks to create novel intellectual property for commercial license.

Special thanks to Rich Rowan and Verena Achenbach for their work on this project.

PROJECT PART-FINANCED BY THE EUROPEAN UNION

Europe and Scotland
Making it **work together**

7. REFERENCES

[1] S. Aggarwal, H. Banavar, A. Khandelwal, S. Mukherjee, and S. Rangarajan. Accuracy in dead-reckoning based distributed multi-player games. In *NETGAMES'04*, pages 161–165. ACM, 2004.

[2] J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang*. Prentice Hall, 1996.

[3] M. A. Bassiouni, M.-H. Chiu, M. L. Loper, M. Garnsey, and J. Williams. Performance and reliability analysis of relevance filtering for scalable distributed interactive simulation. *ACM Trans. Model. Comput. Simul.*, 7(3):293–331, 1997.

[4] A. R. Bharambe, J. Pang, and S. Seshan. Colyseus: A distributed architecture for online multiplayer games. In *NSDI'06*. USENIX, 2006.

[5] J.-S. Boulanger, J. Kienzle, and C. Verbrugge. Comparing interest management algorithms for massively multiplayer games. In *NETGAMES'06*, page 6. ACM, 2006.

[6] E. Cronin, B. Filstrup, A. Kurc, and S. Jamin. An efficient synchronization mechanism for mirrored game architectures. In *NETGAMES'02*, pages 67–73. ACM, 2002.

[7] C. GauthierDickey, D. Zappala, V. M. Lo, and J. Marr. Low latency and cheat-proof event ordering for peer-to-peer games. In *NOSSDAV'04*, pages 134–139. ACM, 2004.

[8] K. Guo, S. Mukherjee, S. Rangarajan, and S. Paul. A fair message exchange framework for distributed multi-player games. In *NETGAMES'03*, pages 29–41. ACM, 2003.

[9] C. Gutwin, C. Fedak, M. Watson, J. Dyck, and T. Bell. Improving network efficiency in real-time groupware with general message compression. In *CSCW'06*, pages 119–128. ACM, 2006.

[10] M. Macedonia, M. Zyda, D. Pratt, P. Barham, and S. Zeswitz. NPSNET: A network software architecture for large-scale virtual environment. *Presence*, 3(4):265–287, 1994.

[11] M. Mauve, J. Vogel, V. Hilt, and W. Effelsberg. Local-lag and timewarp: providing consistency for replicated continuous applications. *IEEE Transactions on Multimedia*, 6(1):47–57, 2004.

[12] Netstat Live. `http://www.analogx.com/`.

[13] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Patterns for Concurrent and Networked Objects*. John Wiley & Sons, 2000.

[14] J. Smed and H. Hakonen. *Algorithms and Networking for Computer Games*. John Wiley & Sons, 2006.

[15] Stackless Python. `http://www.stackless.com/`.