Lessons learned from using a networked entity system in game development

Abstract

In this work we analyze the usage of a previously presented networked entity system for developing multiplayer games. The source codes of the example cases are studied to observe the entity system usage in practice. The main goal is to evaluate the entity system on a conceptual level: does it succeed to provide a productive API for networked game development? What enhancements and supporting functionality could further aid application programming? (XXX NOTE: put the answers / conclusions to here when they exist, questions (also) to introduction?)

The entity system is implemented in the open source Tundra SDK which is used in the evaluation. Observations from game codes are divided into conceptual level entity system remarks, and issues related to the concrete particular implementation in Tundra currently.

Introduction

Networked application programming is generally more complex than standalone software development. The developer typically needs to deal with events, conflicts and error conditions originating from other parts of the distributed system as well as the local user. This is emphasized in multiuser real-time systems compared to the relatively leisurely request-response interaction patterns of most client-server applications.

Higher level abstractions in software are a common way to attempt to ease application development. Regarding networking, libraries exist to simplify creating connections and serializing messages etc. On a even higher level, distributed object systems automate remote calls and data synchronization. For an application developer, these systems are provided as a set of abstractions forming the application development interface (API). However, it is argued that all such abstractions have failure points [leaky-abstractions]. Also it is noted how making good APIs is hard -- creating a bad one is easy [api-matters]. Even a small quirk in an API can accumulate to substantial problems in larger bodies of application code. API design has a significant impact on software quality, and increased API complexity is associated with increased software failure rate [cmu-api_failures].

An entity system for networked application development has been put forth in [Alatalo2011]. This article draws lessons learned from how that system has been put to practice in a few different game development projects. The analysis is done by studying the source codes of the games, and by describing issues encountered in the development. The purpose is to identify leakage points in the abstractions in that entity system and propose areas for improvement.

How can a conceptual design of an entity system be really evaluated? How can we know how well a platform supports actual networked game development? These are not easy questions, but the answers would really help us concretely in game and platform development. We do not claim to provide final answers here. The area of software and API complexity analysis has however made interesting progress recently [TLS paper, the others too?]. (XXX are we lame and: after the initial qualitative ad-hoc analysis here, we review possibilities for more analysis in the future with some of the metrics proposed in the literature -- or can we rock and already apply the techniques here?)

The rest of the article is organized as follows: As background information, we first we give a brief overview of the networked entity system in question, and of the implementation of it in the Tundra SDK. Also, we review existing techniques for API usability research and API complexity analysis. (also?: networked programming libs? alternative models too such as the messaging in sirikata? -- suggest same analysis for sirikata data (kata pong!) as future work?).

As the contribution from this research, a set of example projects are studied. Conclusions are drawn from the observations both to evaluate the conceptual level entity model, and to identify possible improvements and desireable support functionality to be developed in the surrounding platform.

Background and related work

The Entity-Component-Action (ECA) model

- entity system desc (copy-paste from original draft, short, refer to IEEE IC)
    - nice new diagram perhap, to summarize ECA+A

Tundra SDK overview

- what is there besides entities in the Tundra API -- and what is Tundra overall (again hopefully copy-paste from 1st draft)
- other platforms -- Unity3D having the same model etc?

The research methodology - of API complexity research

Recently, software complexity analysis techniques have been applied to statistical (quantitative) studies of API complexity. [cmu-api_failures] studies 2 large corporate software projects and 9 open source projects and finds a link between API complexity and increases in failure proneness of the software. The masses of source code are quantified with measures such as API size and dispersion. Building on existing work (NOTE: Baudi), API complexity is calculated simply from the number of public methods and attributes. In the discussion it is noted how this is severely limited: for example, it fails to take into account pre- and post-invocation assumptions of the API and possibly required sequences of invocation [cmu-api_failures].

A study of 4 alternative implementations, on different frameworks, of the same application uses Object-Points (OP) analysis to quantify the code bases for the comparison [api-complexity-analysis]. OP has originally been developed for estimating development effort, but there the authors adopt it to calculate the complexity of existing software for complexity comparisons. Number of classes, their members and operation calls are counted and assigned adjustment weights in the calculation. Intermediate UML models are used as the data source which allows comparing programs in different languages [api-complexity-analysis].

For the purposes of this study, we do not have enough data for meaningful statistical analysis. Also simplistic measures, suitable for analyzing large bodies of source code, would miss the subtle issues which raise in networked programming on a framework which attempts to hide the intricacies of networking from the application developer. The more fine grained OP analysis, however, is applicable. It does not capture all the elements of API complexity, but would give useful metrics for comparisons. What is missing in our case are the alternative implementations. Therefore we suggest developing the same set of example multiplayer games on alternative platforms for OP and other analysis as future work. OP analysis of one of the examples here, Pong, is given as a starting point.

For the analysis here, we resort to an anecdotal qualitative study. Illustrative examples from various cases are selected to point out strong and weak points in the API. We think that only with qualitative analysis it is possible to get to the heart of the matter: where does hiding networking from the multiplayer game developer succeed and fail with this particular entity system? However we try to lay ground for more rigorous and also quantitative studies in future work as well.

Tundra game projects dissected

- pong and chesapeake / plankton case still
    - concepts of local/remote
        - physics authorativity in Pong
        - the whole net load avoiding, described with the diag in plankton case

The EC system allows for two ways of implementing coordinated entity behaviour accross the server and clients. The first, simpler way is to rely on the the automatic synchronisation mechanism for replicating

changes to entities as they are made. The second way is to handle state and behaviour replication manually using the "entity actions", an messaging mechanism that lets entity-associated scripts to send events to each other. The latter way can be used to reduce the amount of network traffic.

- circus -- what about it? there is gameobject.js and gamestate.js
- mixed reality city game, with websockets too
    - an example where the extensibility seems to work (Q: is there state sync, what does 'addPolice' action actually do etc? - how is the Graffity info in the system?
- others?

Results

The extensibility of the entity system, argued for in the original publication, gets some further support here. A number of quite different applications have been developed using the system, and it has been straightforward to implement the new functionalities without touching the core.

Ease of development shows promise, but would benefit both from API improvements and underlying support functionality such as improved scalability by way of using clever interest management techniques.

For more reliable studies in the future, it would be interesting to apply rigorous software complexity analysis techniques such as Object-Points analysis to comparable codebases. This could be done both to evaluate alternative approaches in different frameworks, such as attribute synchronization in Tundra vs. custom messaging in Sirikata, and to evaluate improvements over time while enhancing a single framework. The comparative analysis would require the same example game, or probably a set of games, to be implemented on all the platforms. We propose Pong as the minimal realtime multiplayer game, but a few representative much more complex cases should be added to the set for meaningful evaluations.

References

api-matters	Michi Henning, API Design Matters, Communications of the ACM Vol. 52 No. 5 http://cacm.acm.org/magazines/2009/5/24646-api-design-matters/fulltext
cmu-api_failures(1, 2, 3)	Marcelo Cataldo1, Cleidson R.B. de Souza2 (2011). The Impact of API Complexity on Failures: An Empirical Analysis of Proprietary and Open Source Software Systems. http://reports-archive.adm.cs.cmu.edu/anon/isr2011/CMU-ISR-11-106.pdf
api-complexity-analysis	Comparing Complexity of API Designs: An Exploratory Experiment on DSL-based Framework Integration. http://www.sba-research.org/wp-content/uploads/publications/gpce11.pdf
leaky-abstractions	http://www.joelonsoftware.com/articles/LeakyAbstractions.html