A comparison of networked game development APIs

or: An analysis of a networked game development API

objective: analyze API complexity of networked game development platform(s) method: OP data: two implementations of pong (and?) results: Tundra is simple, other is more complex?

other concerns: leakages?

Introduction

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).

It has been noted how making good APIs is hard -- and that 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]. Developed in the open source Tundra SDK, it strives to apply best practices from game engine design literature, notably the aggregation using entity-component model. Specifically for networking, it features attribute autosynchronization, a simple form of transparent remote procedure calls (entity actions) and efficient customized movement messages with inter- and extrapolation logic (dead reckoning). The purpose of the abstract entity model, and the whole concrete platform, is to make multiplayer game development easy and productive. The goal in this study is to evaluate whether and how the Tundra API, and with it a a few common practices in modern game and networking libraries, succeed in that.

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 to all of it here. The area of software and API complexity analysis has however made interesting progress recently [api-complexity-analysis] [cmu-api_failures]. By applying a software complexity analysis technique, we investigate one particular aspect of the quality of networked application platforms: the API complexity for a networked game developer.

We analyze API complexity by borrowing an approach from a previous study in a slightly different field. We conduct a comparative study of two alternative APIs for networked game development by analyzing the complexity of the same game implemented on the two platforms. The game is Pong, which is proposed as a minimal hello-world style example of a multiplayer game.

TODO: FINISH

Background

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 gives useful metrics for comparisons.

The game of Pong

TODO

- has been used in networked games before [pong-ping].

Platforms: realXtend Tundra SDK and Union Platform

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

Application of Object-Point analysis

The chosen Sneed's Object-Point (OP) analysis was conducted by automating the collection of most of the key data to derive the variables in the equation. We apply the technique following what has been used for API complexity analysis before in [api-complexity-analysis]

To read the *static class data* for the **Class Points** (CP), we utilize existing source code parsing and annotation systems in API documentation tools. The first alternative implementations of a minimal networked game on different modern high-level APIs studied here are written as a a) Javascript application and b) a combination of Actionscript (as3) for the client and Java for the server module. We developed parsers for the internal / intermediate representation of class and method signatures of JsDoc JSON and AsDoc XML. (The single Java class for b) server we may analyze manually). The class information is read in a Python application to an internal model which contains the data for the Sneed points calculation, implemented in another module in the same Python application.

For the *dynamic function call* information, to calculate the **Message Points** (MP) in the overall OP analysis, we use the Closure Javascript compiler to traverse the source code to collect function calls and their argument counts. Basic filtering with AWK is used to filter in the relevant information from the Closure tree. To be able to analyze also Actionscript code, we do text processing to strip AS extensions to the basic ECMA/Javascript (remove public/private definitions and type declarations). A simple parser made with Python is used to read the function call data required to calculate MPs. This completes the automated data collection and processing developed for the OP calculations here.

The software to run the calculations, together with the datasets used in the analysis here, is available from https://github.com/realXtend/doc/tree/master/netgames/tools/ (pointcounter.py is the executable, with the formula for OP = CP + MP).

TODO: add the equation + legend here

Results

Full applications

Tundra PongMultiplayer

game.js: 74 OP (CP + MP)

UnionPlatform Pong tutorial

client 14x .as3: 273 OP (CP + MP)

- UnionPong/Java/PongRoomModule.java

Only the networking code

- Selected classes, explain the criteria.

TODO

Discussion

First and foremost, we are comparing apples and oranges here: The Javascript game is a script running on the Tundra platform. The UnionPong is a new client application, to which the networking has been added by using Union's Reaktor Flash library.

The Tundra game utilizes a complete otherwise made scene where the pong game takes place. It runs on an existing client-server system, and utilizes several default components from the platform: notably all the data for the appearance and spatial instancing. In contrast, UnionPong not only has code to create the appearance of the game court (as it is called in Court.js), but also to define what data is required for a spatial moving object (PongObject has x, y, direction, speed, width and height). Tundra, again, has the position in the Placeable component and the size and shape information for collisions, and the speed vector for movement, in the physics module's Rigidbody component. So it is no wonder that the Onion implementation gets much higher points, due to having much more of the implementation in the game/application code. Also with networking there is a great difference: OnionPong sends own custom movement messages for all the movement, and has also custom server side code to do ball bouncing, whereas on Tundra the default movement replication and physics collisions are used.

How should we interpret this result? There are several things to consider, these are visited in the following: 1. validity of the analysis technique, the automated (partial) Object-Point analysis 2. nature, suitability and use of scripting vs. application development libraries 3. observations of the high-level network programming APIs studied here (XXX NOTE: both have autosync, heavy use of callbacks, .. - introduce this in some background info thing?) 4. limitations: the many areas of analysis outside the focus here (scalability, efficiency of the networking etc)

1. Validity of the analysis

We apply Sneed's Object-Point analysis, following how it has been adopted to API complexity evaluation in [api-complexity-analysis], as closely as we could with the automated source code analysis. The validity must thus be evaluated from two viewpoints: a) applicability of OPs to API complexity analysis in general and b) the deviations from the intended calculation due to limits of the analysis software.

The OP sums of the full examples have an order of magnitude (right? XXX) sized difference in the proposed complexity of the two implementations of the same game. Noting the aforementioned substantial difference in the nature and scope of the implementations, the ratio of 74:273 (XXX fix when nums update) seems correct for codebases of 2 sizeable and 14(+1) mostly small classes respectively.

TODO: what was left out from analysis (was anything, in the end? XXX)

2. On scripting vs own client development

TODO

- as the data points out, implementing something on an existing platform can be comparatively very little work
- making an own application (client) is easily powerful and straightforward for own custom things, however
- same existing modules/components can be used either way, though. still simpler when don't need to deal with application init and connecting etc.
- does the complexity lurk somewhere still?

3. Observations of the high-level network programming APIs

The APIs under study here are very similar regarding the networking. They both have an abstract container for the state: a Room in Union, and a Scene in Tundra. Application can put own custom state information as special attributes in that container, and the system takes care of automatically synchronizing changes to that data.

Both use callbacks heavily, for example both to listen to new clients entering the service (an event of Room in Union's Reaktor and in the RoomModule on the Union server separately, an event of the Server core API object in Tundra on server side) and to attribute changes coming in over the network.

They both also allow sending simple ad-hoc custom messages, which the Tundra version uses for game events such as informing of a victory (with the associated data), and UnionPong uses for all networking (also paddle and ball movements).

With this in mind, we would expect the difference in the complexity sum derive from the scope of the implementations used in the analysis.

TODO: return to this when the numbers from network-code-only analysis are in too?!?

4. Limitations

the many areas of analysis outside the focus here (scalability, efficiency of the networking etc)

TODO

Conclusions

TODO

(We are happy and curious about using this tool for many kinds of comparisons: longitudal studies of a single API over time, comparisons of e.g. networking stacks when using different protocols for similar functionality, ... or?)

Similarities and differences of using a platform as ready made client software, on which just run scripts, vs. libraries to create own applications, are interesting to study more. Same software components (libraries, modules etc) can be used in both configurations -- what is more suitable may well depend on the particular case.

(XXX Q: where does complexity lurk? should we consider the leaks here? does Onion have something to handle them? at least had the Attribute setting exception in the java server XXX)

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, 4)	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
pong-ping	High and Low Ping and the Game of Pong. http://www.cs.umu.se/~greger/pong.pdf