Modular Decomposition in Heterogeneous Environments:

Decoupling Packaging Boundaries from System Architecture

William Foote

June 7, 2012

Effective decomposition of software systems into modules is considered. The common practice of splitting the top-level modules of a system architecture along software packaging boundaries, such as between client and server in a distributed system, is analyzed. The modular decomposition thus obtained is evaluated according to the criteria articulated by Parnas, namely, loose coupling and high cohesion. Decomposition along packaging boundaries is found to result in fragile modules that exhibit tight coupling along the inter-module boundary, and lower cohesion within the modules concerned. The inter-module boundary is found to be frequently subject to change in a heterogeneous environment, because such boundaries are often performance-critical.

An alternate modularization scheme is proposed, where the "bridge" between two or more software packages (e.g. executables) is called out as a top-level module in its own right. This module contains some code that runs within each packaged unit of software (e.g. within the client executable and within the server executable), and encapsulates the protocol between them. The interface between this module and each of its peers can be expressed as an in-process API, which is less subject to change for reasons of performance optimization or other reasons not intrinsic to the problem domain. It is argued that this high-level design scheme results in a more effective system architecture, leading to lower cost, superior time to market, and better maintainability and performance.

Contents

1 Introduction

Software systems have dramatically increased in size, scope and complexity over the lifespan of our comparatively young field. Indeed, it is often remarked that things have changed so quickly and dramatically that we sometimes fail to fully appreciate the value of what has come before us. Many software systems today are large, brittle, hard to maintain, and hard to extend. They are worked on by very large, distributed teams. The amount of code in these systems is orders of magnitude larger than systems of the past, and the hardware is much faster, but many of the engineering challenges are strikingly similar to large systems of the past. An effective modular decomposition of a software system can help. We analyze the shortcomings of a typical decomposition of a software system along packaging boundaries. We then propose an alternate approach, and argue that it results in higher cohesion and lower coupling, leading to lower costs, greater flexibility and greater efficiency.

1.1 Parnas' Criteria

In 1972, David Parnas articulated a view of what constitutes a good module in software design [1], centering around the ideas of loose coupling and high cohesion. He advocated for a design that maximizes information hiding, where "information" is defined broadly. The information that is to be hidden includes internal data structures, complex or changeable design assumptions and arbitrary decisions. His work was highly influential in the discipline of Object-Oriented analysis and design and OO programming languages.

It has been observed that Parnas' conception of information hiding is at times conflated with simple data hiding. For example, in [2], Rogers states, "Authors rarely distinguish between [encapsulation and information hiding] and often directly claim they are the same," where encapsulation is taken to mean the OO language mechanism of the "bundling of data with the methods that operate on that data." He argues, "protecting internal data is only one of the many concerns driving design decisions ...." Like Parnas, he emphasizes the value of planning for change, and isolating the scope of a potential change within a boundary. Rogers demonstrates that isolating change involves more than just data hiding, even for a relatively small Java class.

Heterogeneous systems consisting of multiple executable units present a particular challenge to a modular system architecture. A heterogeneous system features natural software packaging boundaries between the platform components that make up the system. These software packaging boundaries are tempting places to draw the module boundaries in a system architecture. For example, in a client-server system, it is tempting to view the client as one module, and the server as another module. Of course, further modular decomposition of each of these large modules can be done, but the top-level modular decomposition often follows the packaging boundaries imposed by the network topology, choice of tools, when the

computation is carried out (e.g. at build time, initialization or run time), or by other criteria that are not intrinsic to the problem domain.

1.2 Effective Modular Decomposition

In this paper, we argue that decomposing systems into modules along packaging boundaries often results in systems that exhibit higher coupling and lower cohesion than would otherwise be the case. As such, these systems are fragile, expensive to maintain, slower and more complex than they need be. Further, their design is not what Parnas calls "beautiful" in [3], because they do not encapsulate changeable or arbitrary information.

Instead, we propose a broad definition of what constitutes a module, to wit:

> A module is a collection of one or more software artifacts into a cohesive whole that has defined behavior and defined interfaces to other modules.

This definition says nothing about software packaging. A module is viewed as essentially a mental model of a part of a system, where that part is not constrained to exist within any particular unit of packaging. In section 4, we explore the definition of a module in greater depth.

We begin by analyzing two systems. First, we explore a client-server system for selecting services and displaying structured text. We examine the tradeoffs of a traditional modular decomposition that has a boundary between the client and the server, and an alternate design that encapsulates this boundary within a "bridge" module. Next, we explore a scene graph implementation that is designed to hide the distinction between computation carried out at build-time and runtime, enabling transparent apportioning of work as is most appropriate for a particular deployment scenario.

Following this analysis, we discuss some of the challenges of drawing module boundaries as we suggest, and explore reasons why this is often not done. We then present the case that despite the challenges, superior results can be achieved by concentrating on a good modular composition that maximizes cohesiveness and minimizes coupling, even when this means having modules that cross software packaging boundaries.

2 Client/Server System

2.1 Background

In our first example, we consider a system for displaying lists of services including structured text that is stored on a server. We wish to display this text on a device that presents special

challenges: A Blu-ray player connected to a TV set. This particular example of structured text was chosen to illustrate concepts that were observed in multiple commercially deployed systems. The pattern illustrated with this example has been observed in a very large connected system that targets Android phones and tablets, and also in a commercial application of connected Blu-ray players. From a design perspective, both Android and Blu-ray presents constraints that are an interesting challenge to the systems designer. Blu-ray players in particular have a slow CPU, limited memory, and a large, 1920x1080 pixel screen; these result in a different design space than is the case for a distributed system using a web browser as a client. For this reason, our small example discusses Blu-ray, even though the same phenomenon has been observed elsewhere.

The example shown here was first presented in [4].

We consider a connected application implemented using Java Server Pages (JSPs) on an application server. It connects to an external social media application, and it stores data on a separate database server. It supports two kinds of clients: PCs running a web browser, and a connected Blu-ray application. Static assets are served via a CDN. A typical network topology for such a system is pictured in figure 1.

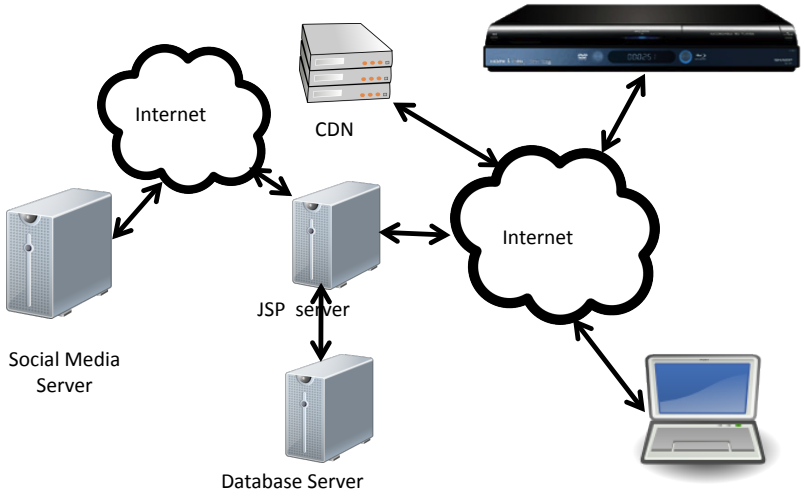

Figure 1: Network Topology

We concentrate on the application as experienced by a user whose client device is the Blu-ray player. As previously discussed, this environment helps to expose inefficiencies and other problems in the system architecture. A tempting architecture for such a system is shown in figure 2 on the next page.

The major system modules have their boundaries mostly along physical machine boundaries. In addition, within the Blu-ray application, another component boundary is drawn

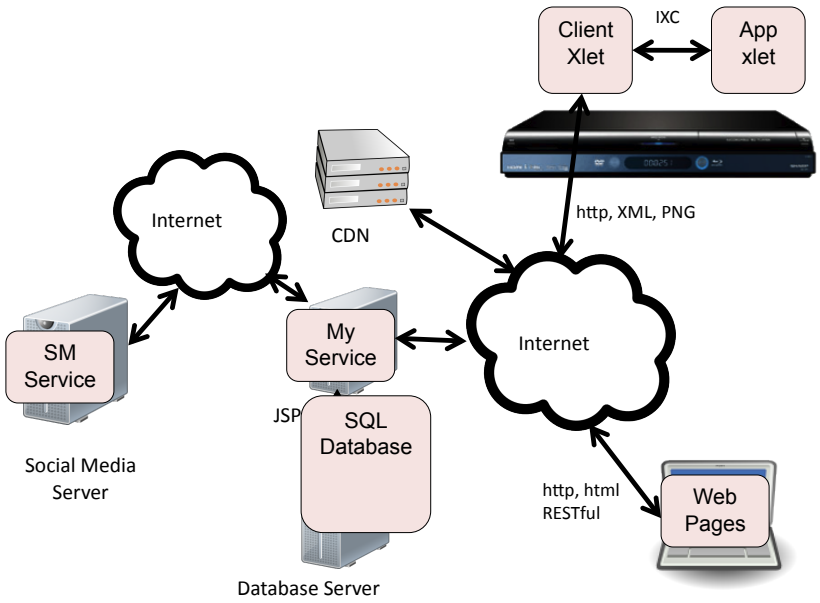

Figure 2: Topology Driven Architecture

along software packaging boundaries, with Blu-ray's inter-xlet communication providing the channel between them. In Blu-ray, applications are packaged as "xlets," and two concurrently executing xlets can communicate with a remote procedure call mechanism called "inter-xlet communication" (IXC). In figure 2, most of the module boundaries lie along the "natural" boundaries imposed by the topology of the communications channels. They also all lie along boundaries that reflect the packaging of executable software components. One exception is the SQL database. Many database implementations present an API to the applications programmer, thus hiding the communications mechanism used between the application server and the database server.

We argue that superior results can be achieved by structuring the parts of the system that are under one's control in the same way as the database mentioned above. That is, the communications boundary, and potentially other boundaries imposed by the packaging of software executables can be logically encapsulated within a module, where this module interacts with other modules via in-process APIs, rather than a communications protocol. In effect, a key module in this new architecture is an adapter that interacts with the JSP service via an API on the application server, and interacts with the Blu-ray application running on the player via an API. The details of communication between the two physical devices is logically encapsulated within this "bridge" module. This is pictured in figure 3 on the following page. The "BD-Live Adapter" component includes code that runs on the server, and code that runs on the client. It further includes knowledge of data stored in a Content Delivery Network

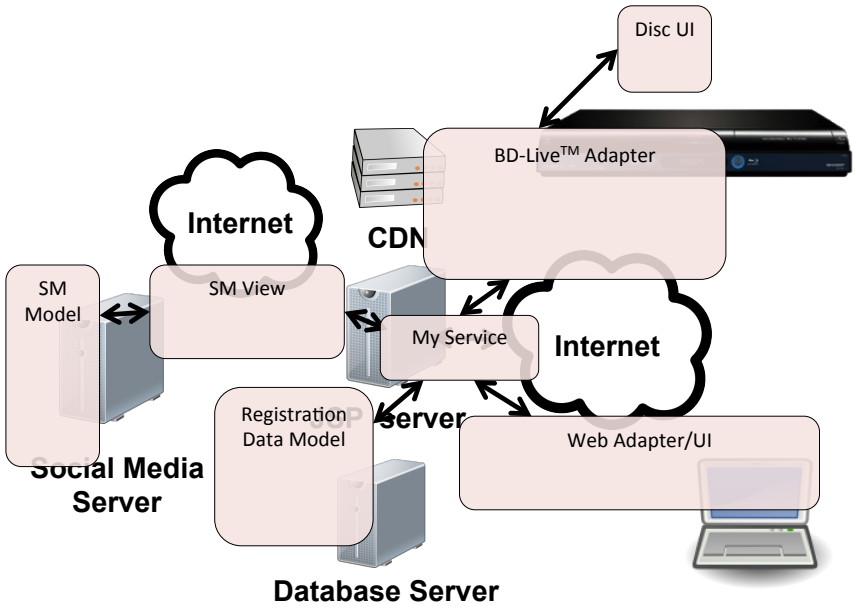

Figure 3: Adapter Modules Encapsulating Communication

(CDN).

2.2 Geolocation of a list of services

As a first example scenario, we explore obtaining a list of available enhancements, or services, from a server. We assume that this list changes based on geolocation. That is, the IP address from which the request originates is used to determine the region where the viewer is located. Based on the region, a different list is presented. This kind of distinction could be made, for example, in the list of movie previews that should be shown to a viewer. This list generally varies by the country or other region, due to differing movie release dates, marketing strategies, etc. We first explore two top-level designs. Next, we measure the performance improvement of an optimized version of the application, and explore the cost of this change to the system resulting from each of the two designs. Then, we explore the impact of a change in the application's functionality.

2.2.1 Top-level Design Alternatives

A traditional top-level decomposition of this system along executable packaging lines would involve two top-level modules: The JSP application, and the Blu-ray application. Often these

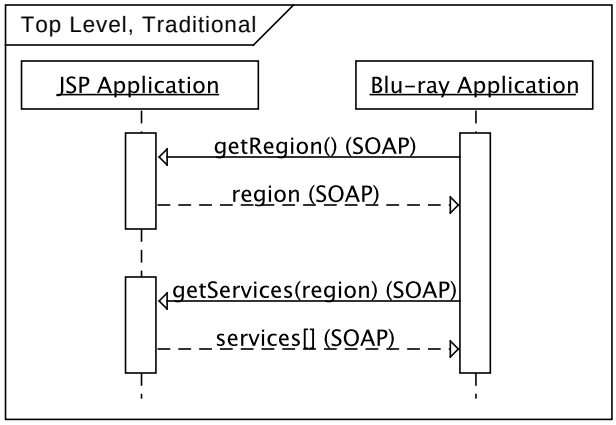

Figure 4: Geo-located Services, Traditional Top-level Module Structure

modules would be assigned to different developers, one who is deeply familiar with server-side programming, and another well versed in the nuances of Blu-ray programming. Between them, the module interface would consist of some kind of networking protocol or remote procedure call (RPC) mechanism, e.g. SOAP. To support the "get list" functionality, it would be natural to break this up into two calls: `getRegion()` and `getServices(region)`. Indeed, `getRegion()` would likely already exist, so using this method would be taken as an example of desirable code reuse. An interaction diagram illustrating the resulting top-level architecture is pictured in figure 4.

Typically, the top-level modules of a system like this correspond to different developer roles, and often correspond to different individuals, or teams. In a larger system, this top-level module structure can have a profound effect on organizational structure. The interface between top-level modules becomes the point of interaction between two individuals, or two teams. This results in the communications protocol between the modules becoming entrenched, and hard to change, because changing it involves generating consensus between at least two groups within the organization. This is unfortunate, because this communications protocol is subject to change, e.g. because network bandwidth and latency are precious resources whose optimization later in the development cycle is highly desirable.

Consider, next, a different top-level module structure. Here, we call out as a top-level module the "bridge" between the application logic resident in the application server and the display-centric logic resident in the Blu-ray application. From the perspective of the developer(s) of the bridge module, the resulting system and its interactions are as pictured in figure 5 on the following page.

In this design, the interface between the Bridge and the rest of the JSP application is an in-process API, as is the interface between the bridge and the rest of the Blu-ray application. This client API is more likely to be stable, because is written in terms of the app's semantic requirements. For example, getting a list of services is a single API call, with the detail

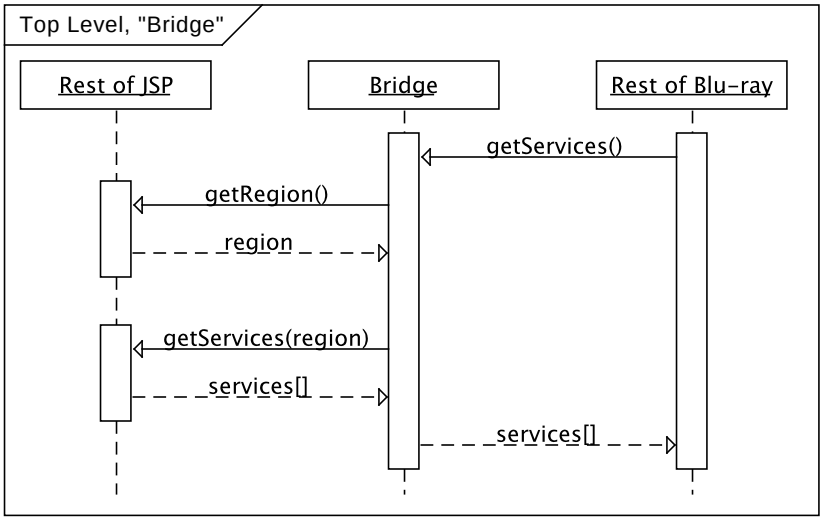

Figure 5: Geo-located Services, Module Structure with Bridge

of geo-location encapsulated within the bridge. Further, as these APIs are in-process APIs, there is not much pressure to optimize them for performance reasons. In-process API calls execute with an overhead on the order of microseconds or less, whereas remote procedure calls execute in a matter of tens or even hundreds of milliseconds.

Of course, when we encapsulate the network protocol within a module, it doesn't disappear; it's simply relegated to the status of module implementation. With the bridge-based system architecture, the internals of the initial version of the bridge module might well use exactly the same SOAP RPC mechanism in its implementation as was present at the top level of the first design. The difference here is that this network protocol is viewed as an implementation detail of the bridge module, and the other top-level modules are considered as external actors. The internal design of the initial version of the bridge module is as pictured in figure 6 on the next page.

The initial version of a system built according to both designs would have nearly identical performance characteristics. Indeed, they might result in identical executable code. This phenomenon was observed by Parnas in [1]; a different modular decomposition effects how the team views the software product, and doesn't necessarily impact the bits that comprise the final executable. The improvement resulting from a better modular decomposition comes in terms of development time, maintainability, reliability, staffing, and other "soft" criteria.

2.2.2 Optimizing the List of Services

Now we consider what happens to the system when we change one of the initial design assumptions. The simple data protocol outlined earlier is completely appropriate for a proto-

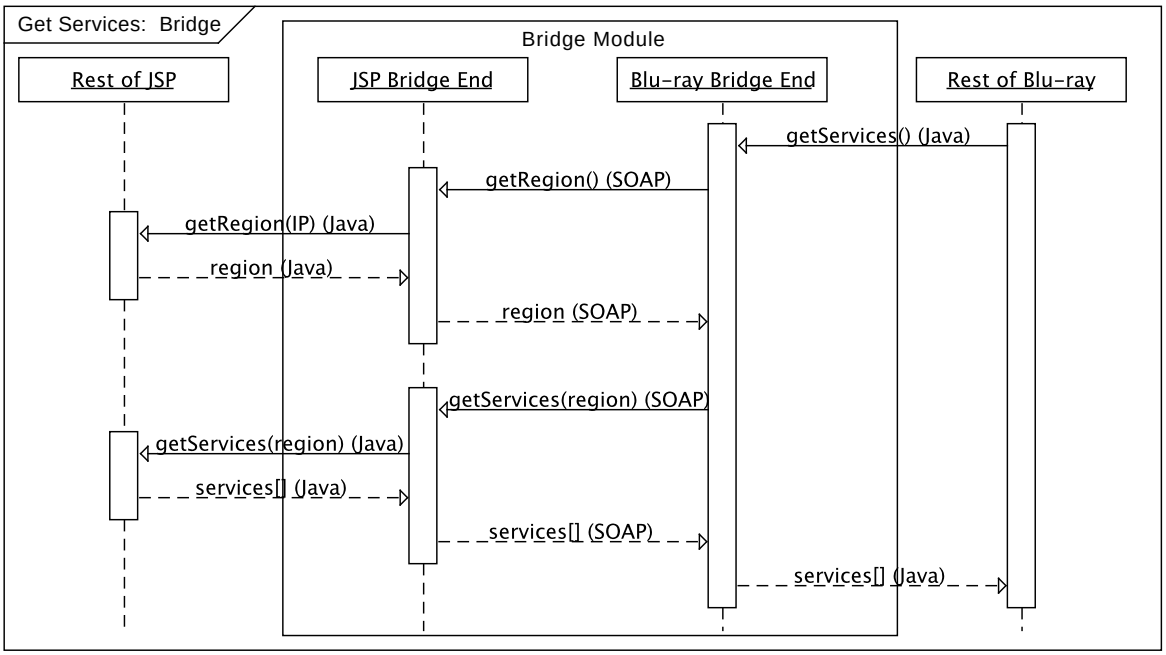

Figure 6: Geo-located Services, Internals of Initial Bridge Module

type, or a system that isn't too widely deployed. However, it is somewhat inefficient. Two server round-trips are required. This increases the latency of the `getServices()` call. If the server is on the other side of the globe, this increased delay to the viewer could be in the hundreds of milliseconds. In addition, this protocol burdens the main application server with the sending of a relatively large asset, the list of services. Optimizations are possible.

In this system, the list of services depends only on the viewer's region. Typically, the number of regions representing linguistic or jurisdictional boundaries would be a small number, perhaps in the hundreds. There is no reason why these regionalized service lists could not be pre-computed, and stored on a content delivery network (CDN). In this realization, the selection of which list of services to send still happens on the main application server, but the bulk data comes from a CDN node that is geographically close to the viewer, and that has good connectivity to the client device. An interaction diagram for this realization of this optimized protocol according to the bridge-based architecture is shown on figure 7 on the following page.

While more complex, this optimization results in a faster and more scalable system. A small overhead is paid to pre-populate the CDN, and to initialize the services table in the application server at startup. Once the system is running, however, the load on the critical application server is small. It need only do a lookup of region based on IP address (which can be effectively cached), and then do a URL lookup in an in-memory table. This level of processing can even be done without a context switch on the server, in the normal case of a cache hit

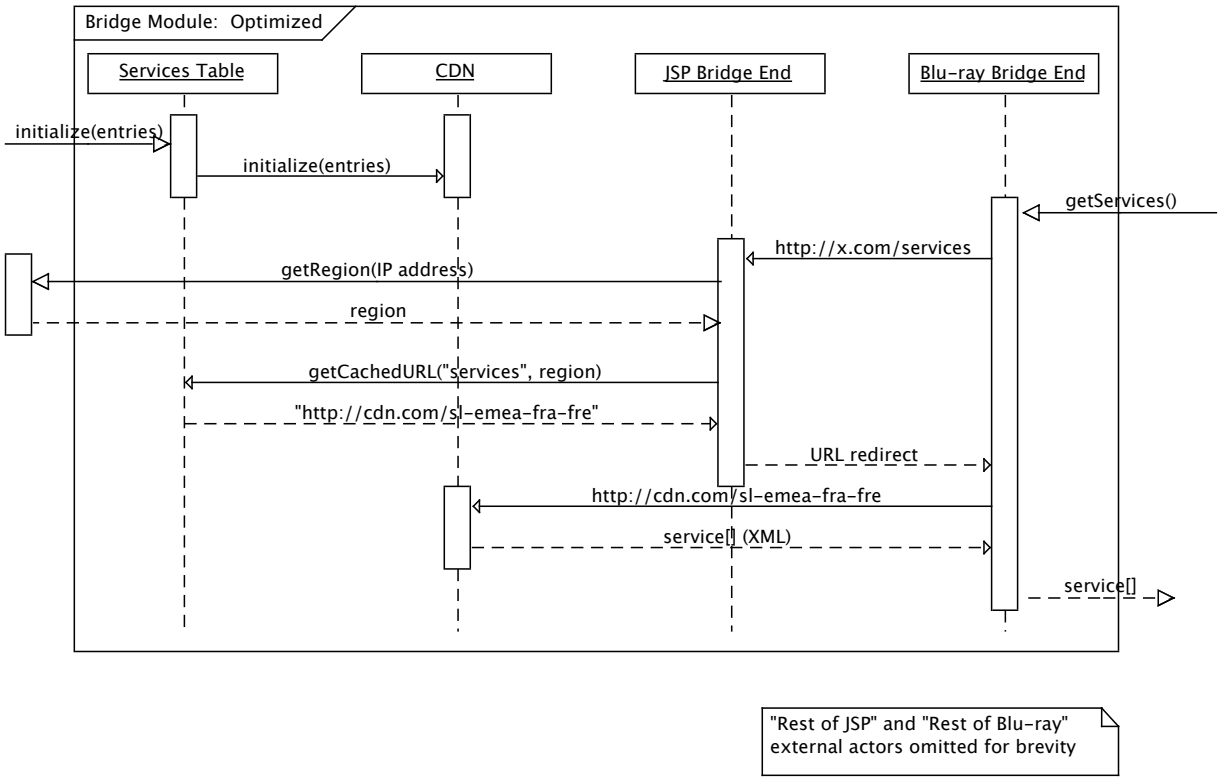

Figure 7: Geo-located Services, Optimized Bridge Module

where the initial query is in one packet. In terms of network traffic, the main server receives one packet (the query), and sends one packet (the URL redirect). All of the bulk data transfer is done from a simple and inexpensive CDN node. This improved protocol should be faster in all cases, because the CDN round-trip should be faster than a round-trip to the server, and can be significantly faster depending on network traffic and distance to the application server. Perhaps more importantly, this optimized protocol results in a more scalable system, because the application server offloads the bulk data transfer to a CDN.

Of course, further optimizations are possible, such as distributing the application server. This first, relatively simple optimization was chosen to illustrate the design implications.

In the bridge-based architecture, this change is entirely contained with one module (the bridge). Optimization of networking performance in this system can be done by a single team that is expert in this area. This one team is required to deliver a small amount of code on both the client and the server device, but is not required to do “heavy lifting” on either. By way of contrast, with the topology-based architecture, the network protocol falls squarely between two teams. In a sense, it is both everybody’s responsibility, and nobody’s. Both teams could well have opinions about what the network protocol should be, and they could well be divergent. The (changeable) network protocol becomes an item of negotiation between two teams, rather than an implementation artifact managed by one team. In other words, with the topology-based architecture, we have a system with two large, tightly-coupled modules

whose interface is subject to change as the system evolves. With the bridge-based architecture, we have three smaller modules with in-process API-based interfaces between them. As shown, these API interfaces are less subject to change, and they can be less complex and thus easier to understand and maintain.

2.2.3 Semantic Change to the List of Services

Next, we consider the impact of a semantic change to the list of services. Consider a change where the service list becomes a function of both the region and of some function based on the identity of the user. For example, the set of movie trailers to be shown might be influenced by both what is available in their region, and by a heuristic function that models what that user might be more likely to buy, based on previous purchases or other user profiling. For this example, we assume that the set of all possible trailer groupings is still of manageable complexity, with most (but not necessarily all) queries resulting in a list that can be pre-computed.

This change introduces state into the transaction. From the user's standpoint, there would be some kind of login state that would maintain a unique credential establishing identity. Different server implementations of this are possible. In one, the server could maintain the state as part of a persistent session. In others, e.g. a RESTful server, the server would be stateless, and the identity credential would be passed along with every relevant transaction, such as the get services call.

Without a bridge module, this difference would need to be reflected throughout the Blu-ray Application. For example, in a RESTful implementation, the (e.g. SOAP) `getServices(region)` query would change to `getServices(region, identity)`. Similar changes would likely be needed throughout the application.

With a bridge module, the change could be almost completely encapsulated within that module. For example, the `getServices()` query would be almost completely unchanged. The only required change would be a potential failure mode in the case where the user had not logged in, if indeed this is not a supported mode. At this level of the system, this could be represented with a new checked exception.

The more significant change with the bridge module would be a new API to establish the login. This could well be a simple stateful mechanism. That is, the bridge module could easily maintain a small amount of state, recording the current login credential. With a session-based server, this could reside on the server (with perhaps a copy on the client for failure recovery), whereas with a RESTful server, the state would be recorded on the client. In both cases, the difference would be completely encapsulated within the bridge module; other parts of the system, both in the client and in the rest of the server system, would not be impacted.

Next, we turn to the question of how the results would be delivered. Note that we said that *most* of the queries would result in pre-computed lists. Thus, for most queries, the optimized implementation of section 2.2.2 would continue to work. But what of the case where it

didn't work? To pick a rather sophisticated example, consider the case where a user's query resulted in the union of two (mostly disjoint) pre-computed sets of services, plus a list of services not in a pre-computed set. This might be, for example, the "US nature lovers" set and the "US parents of children under the age of 7" set, plus two services that are based on previous purchases by that customer.

A reasonable server reply for this query would consist of the URLs of each of the two groups, plus the direct data for the two additional services. For this, a simple URL redirect would no longer work; the wire protocol would need to support sending a structured message. Additionally, the client would likely need to merge the different sources of data, maintaining the sort criteria and/or grouping.

Without a bridge module, this would be a quite disruptive change, involving substantial agreement between two engineering groups. The change in protocol would need to be coordinated in time between the two groups, and testing would need to be arranged. With the bridge module, this enhanced functionality could be implemented by first creating the needed facilities on the back end in a non-disruptive fashion. Once ready, all of the changes needed to accommodate this change would be entirely contained within the bridge module. The physical deployment would require coordination among different systems, but importantly, the responsibility for rolling out the change, testing it, and ensuring a smooth deployment would unambiguously rest with one team: The bridge team.

As can be seen from these examples, well-crafted APIs between the bridge module and other modules in the system are resilient to change. A bridge module with a client component is not constrained by factors unrelated to the problem domain, such as the typical RESTful constraint of not maintaining a session on the server. In effect, the client portion of the bridge allows the rest of the client to use simple session semantics, even when the "session" is implemented solely within the client portion of the bridge. This reduces the fragility of the API offered to the rest of client code, and thereby makes it less costly to introduce semantic changes into the application.

2.3 Parsing and Formatting Structured Text

Next, we turn to a different aspect of the communications protocol between client and server. In the above, the list of services was sent as an XML document. Is it worthwhile sending this data in a more efficient binary format? And, what about moving other computation from the client, either by pre-computing it and storing the results, or performing it on the server? These changes could be accommodated with the bridge-based system with a minimum of change outside the bridge module itself, whereas with the topology-based architecture, changes of this nature would again involve negotiation between multiple teams.

In order to measure the impact of these improvements, we turn to a similar parsing problem where data can be readily obtained. We choose a sample data set consisting of 50K of text

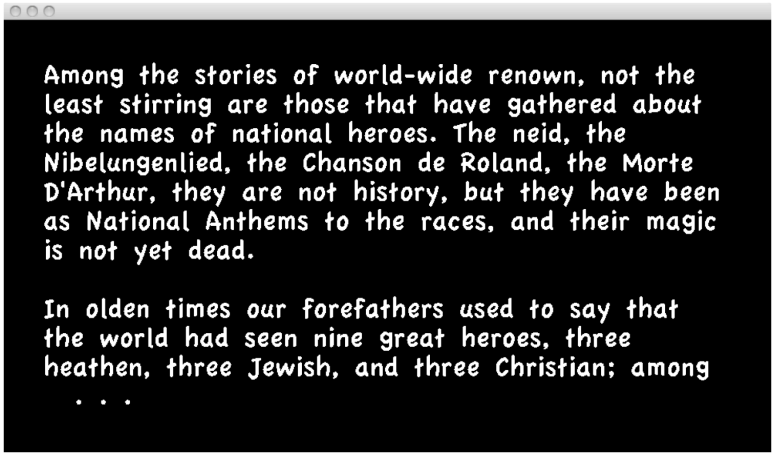

Figure 8: The Song of Roland

formatted into chapters, sections and paragraphs. This is representative of the kind of processing that was observed in deployed commercial systems in other domains. For our sample data, we chose an English translation of The Song of Roland, written in the late 11^{th} century. A screenshot of the application running in a desktop emulator is shown in figure 8. We further measure the effect of pre-computing the text layout, based on font metrics. Again, this is representative of a task that was done on a real, deployed system; the real system did not pre-compute the text layout. This change is something that would most naturally be done during an optimization phase of a project. It would be more likely to happen if it could be accomplished within the boundary of a single module. Indeed, in the commercial system this is modeling, this optimization was not done, and the topology-based architecture was one reason why this was so.

2.3.1 The Application

To measure the performance of different performance optimizations, we built a Blu-ray application that opens a connection to a web server, and reads the data in both XML and binary format. As a further experiment, we also parsed a JSON representation of the data that is logically equivalent to the binary representation. The results were timed on a typical commercial Blu-ray player. The code to open the connection is common to all three:

```
token = Profile.startTimer(profileOpenURL, TID);
URL url = new URL("http://hdcookbook.com/tmp/morte_darthur.xml");
InputStream is = url.openStream();
Reader r = new InputStreamReader(
               new BufferedInputStream(is), "UTF-8");
Profile.stopTimer(token);
```

For the XML implementation, the XML document is first parsed, and then the document tree is visited, converting it to the internal format needed by the application.

```
token = Profile.startTimer(profileReadXML, TID);
Parser parser = new Parser();
Element e = parser.parse(r);
r.close();
Profile.stopTimer(token);

token = Profile.startTimer(profileFormatXML);
ArrayList chapList = new ArrayList();
visitXMLGraph(e, chapList);
Profile.stopTimer(token);
```

The XML implementation further formats the data based on font metrics.

```
token = Profile.startTimer(profileFormatXML, TID);
String line = "";
for (;;) {              //  build line
    int pos = remaining.indexOf(' ', 1);
    if (pos == -1) {
        pos = remaining.length();
    }
    String word = remaining.substring(0, pos);
    String newLine = line + word;
    if (font.getStringWidth(newLine) < MAX_WIDTH) {
        line = newLine;
        remaining = remaining.substring(pos);
        ...
    }
}
Profile.stopTimer(token);
```

The JSON implementation is similar in terms of parsing and visiting the document tree. The formatting step is not needed, because in this optimized implementation the data is pre-formatted on the server. Simply pre-formatting the text works in the Blu-ray environment, because the screen dimensions are fixed at 1920x1080, and the font technology used for this sample is all implemented at the application level, guaranteeing identical rendering across players. In other environments, the details of pre-computing the formatting information might be different, but similar schemes are often possible.

In the JSON sample code, we show more of the details of visiting the document tree for illustration.

```
token = Profile.startTimer(profileReadJSON, TID);
```

```
Object[] book = (Object[]) JsonIO.readJSON(rdr);
rdr.close();
Profile.stopTimer(token);

token = Profile.startTimer(profileFormatJSON);
Object[] strings = (Object[]) book[0];
Object[] text = (Object[]) book[1];
Object[] font = (Object[]) book[2];
Chapter[] result = new Chapter[text.length];
for (int i = 0; i < result.length; i++) {
    Chapter chapter = new Chapter();
    result[i] = chapter;
    Object[] textI = (Object[]) text[i];
    Object[] fontI = (Object[]) font[i];
    chapter.pages = new Page[textI.length];
        for (int j = 0; j < textI.length; j++) {
            . . .
        }
    }
}
Profile.stopTimer(token);
```

For the binary implementation, we read the file directly into the needed data structures, so no tree-visiting step is necessary:

```
token = Profile.startTimer(profileReadBinary);
String[] strings = new String[dis.readShort() & 0xffff];
for (int i = 0; i < strings.length; i++) {
    strings[i] = dis.readUTF();
}
Chapter[] result = new Chapter[dis.readShort() & 0xffff];
for (int i = 0; i < result.length; i++) {
    Chapter chapter = new Chapter();
    result[i] = chapter;
          . . .
}
Profile.stopTimer(token);
```

2.3.2 Performance Measurements

The tests were run a total of five times in a loop, to detect the effects of just in time compilers and other variable effects. As expected, the first run of each alternative was slower than

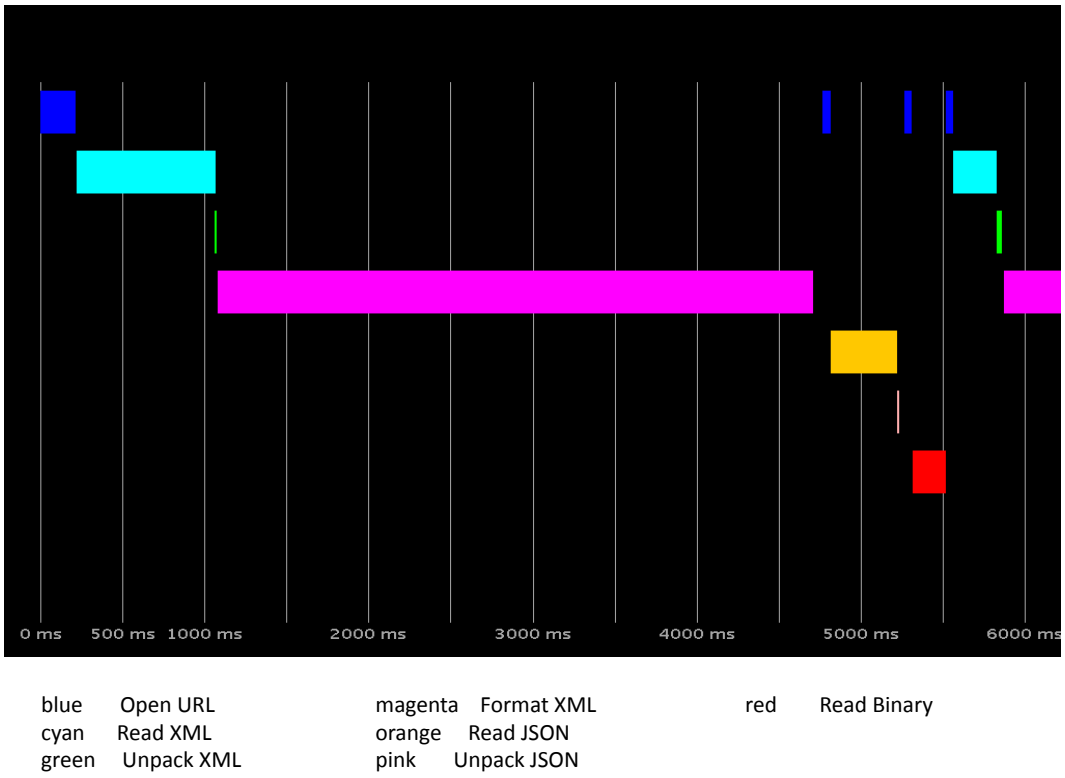

Figure 9: Roland's Parsing Performance

subsequent runs, but they differed in interesting ways. This data was collected on a popular hardware Blu-ray player connected to the internet via a consumer cable modem to an average web server about 1000 miles away. The performance results from the beginning of the run are shown graphically in Figure 9.

In the following table, we present the median execution time of each of the steps over the five sequential runs. In order to highlight the effect of formatting, we also present a row for XML with the formatting step omitted.

Median Execution	Open URL	Read and Parse	Unpack	Format	Total
XML	43 ms	278 ms	8 ms	3303 ms	3632 ms
XML w/o format	43 ms	278 ms	8 ms	-	329 ms
JSON	43 ms	228 ms	7 ms	-	278 ms
Binary	43 ms	151 ms	-	-	194 ms

The results for the first run of each alternative are shown next. It may seem counter-intuitive, but these results may be the more interesting ones to consider. The first execution run is more representative of the case where structured data is read at application start-up time. This is indeed representative of an important class of real-world applications.

First Execution	Open URL	Read and Parse	Unpack	Format	Total
XML	216 ms	845 ms	16 ms	3628 ms	4705 ms
XML w/o format	216 ms	845 ms	16 ms	-	1077 ms
JSON	216 ms	404 ms	14 ms	-	634 ms
Binary	216 ms	202 ms	-	-	418 ms

2.3.3 Analysis of Results

Considering the first execution times, we see that the optimization results in a speedup by a factor of 11. This is a full order of magnitude of performance increase, from 4.7 seconds to 0.4 seconds. In an interactive consumer-facing application, a four second speedup is commercially relevant.

The change from XML to binary is only responsible for a speedup by a factor of 2.6, resulting in the application starting 0.7 seconds faster. This is still worth doing, but not as dramatic. In fairness, only the change from XML to binary is completely contained within the "bridge" module. The more dramatic 3.6 second speedup is obtained by pre-computing the formatting (or, perhaps, computing it on the server, which can be expected to be at least one or two orders of magnitude faster than the client for this kind of computation). This change likely would result in changes to other modules. Notably, the formatting sub-module's code would move from the client executable (probably within a higher level UI module) to the server, and the API interface between the bridge and the UI module would need to change, to reflect the fact that formatted text now flows across that boundary.

The bridge-based architecture is not a panacea. However, even though there still is a coupling between these two client modules, an API module contract is easier to change than a network protocol, because the details of the API are less performance-critical, and because the two modules have a common frame of reference (namely, the platform on which they execute). Thus, we believe that it is fair to say that the bridge-based architecture removes a structural impediment to the optimization of pre-formatting the text.

Turning now to the median execution time over five runs, we see a similar pattern. Interestingly, the speedup from the switch from XML to binary is now only a factor of 1.7, as opposed to being a factor of 2.6 for the first run. This perhaps reflects the effect of the just-in-time compiler (JIT), which is able to optimize away more of the overhead associated with the XML parser used. This parser consisted of approximately 10 classes, whereas the binary file reader was written in very compact code all contained within one class.

Once formatting is included, the speed up for the optimized version increases to a factor of 19, versus a factor of 11 for the first run. This is perhaps because font metrics calculations are relatively simple integer operations in our application-level font implementation, and are thus less amenable to JIT speedups. The total speedup of the optimized version is 3.4 seconds, which is again commercially relevant.

Given these results, we see that even for this simple example application, commercially-relevant increases of performance can be achieved by replacing the protocol between distributed elements in a client/server system such as this with a more optimized version. Adding an optimization of this nature is better supported by a system architecture that views the bridge spanning executable units as a top-level module. This supports the contention that the bridge-based architecture leads to better system performance, lower cost, and better maintainability.

2.4 Server-side Bridge Implementation

As mentioned above, the implementation of the server side of the bridge should be relatively simple. It should interact with adjacent modules in the architecture through in-process APIs. Generally speaking, caching, timeouts, performance tuning, retries and the like should not be exposed to the bridge code. The bridge should only receive high-level failure notifications, on the level of "error, retry not recommended at this time" or "error, retry to a different host may be fruitful." Fine-grained timeouts and/or retries of internal subsystems should not be exposed to the bridge programmer.

However, it is inevitable that the parallelism model of the server will have some impact, even for the simple protocol-formatting code. In the following sections, we investigate three alternative implementations of the server side of the bridge, for the `getServices()` query of section 2.2. Other models for dealing with parallelism, e.g. those based on Communicating Sequential Processes (CSP, [5]) are possible as well. In all cases, code dealing with error handing is omitted for brevity.

2.4.1 Synchronous Bridge Code

The simplest approach is to have fully synchronous bridge code that blocks on the calls that it makes, even when those calls involve disk I/O, remote procedure calls, or the like. This simple approach can be adequate for many applications, and achieves good scalability on modern computer systems. The bridge code on the server has no need to maintain state between calls, so it can quite easily be resident on some number of load-balanced servers.

For the purpose of illustration, we assume that the server will call the method `getServiceList(QueryContext)`, where `QueryContext` carries information known to the server, such as the IP address of the requestor. We further assume an API to server-side services, specifically a geolocation service accessed through the variable `myGeolocationService` and a query for the URL of a service list by region, supported by `myServiceListDatabase`. The response is modeled by the interface `QueryResponse`, which is implemented by `URLRedirectQueryResponse`.

The synchronous version of the bridge code is as follows:

```
public QueryResponse getServiceList(QueryContext context) {
    IPAddress address = context.getCallerIP();
    Region region = myGeoLocationService.getRegion(clientIP);
    URL u = myServiceListDatabase.getURLForRegion(region);
    return new URLRedirectQueryResponse(u);
}
```

The call to `getRegion(IPAddress)` is a blocking call. Its implementation might involve a cache lookup, and in the case of a cache miss, there might be an RPC to a geolocation service, with associated timeouts, retries, etc. Similarly, the lookup of the list of services might involve similar activities, or it might be done entirely in RAM, e.g. if the list of service list URLs is small enough to comfortably fit, and be pre-populated. In any case, the implementation and optimization of these API calls is not properly the concern of the bridge programmer.

2.4.2 Futures-Based Bridge Implementation

While adequate for many needs, the purely synchronous bridge implementation does not allow the bridge programmer to exploit potential parallelism. In our simple example, there is no parallelism to exploit, but a more substantial example might have. The `Future<R>` class from `java.util.concurrent` provides a well-known, easy way to express such potential parallelism. In the case where the query is handled locally (such as a cache hit), using a future introduces negligible overhead. The futures-based implementation of this query is used as follows:

```
public QueryResponse getServiceList(QueryContext context) {
    IPAddress address = context.getCallerIP();
    Future<Region> region =
        myGeoLocationService.getRegionFuture(clientIP);
            // do something else potentially
            // time-consuming, during which time the
            // geolocation lookup will happen in parallel
    URL u = myServiceListDatabase.getURLForRegion(region.get());
    return new URLRedirectQueryResponse(u);
}
```

This allows the geolocation lookup to occur while other activities are happening. Note that we do not use a future for the `getURLForRegion()` call, because there is no parallelism to exploit; the result is needed immediately after the query. Here, if the `getURLForRegion()` call deferred creating the result, our code would immediately block on the `Future<URL>` object that would be returned.

2.4.3 Callback-Based Bridge Implementation

A callback-based implementation is guaranteed to never block. Each response happens in the form of a callback. This is good for scalability, because the overhead associated with threads that can block is avoided. The resulting application code can become somewhat dense, however. Our example written in this style is as follows.

```
public void getServiceList(final QueryContext context,
                           final QueryResponseCallback cb) {
    IPAddress address = context.getCallerIP();
    myGeoLocationService.getRegion(clientIP, new GeolocCallback() {
        public void regionArrived(Region region) {
            getServiceList(context, region, cb);
        }
    });
}

private void getServiceList(final QueryContext context,
                            final Region region,
                            final QueryResponseCallback cb) {
    mySerivceListDatabase.getURLForRegion(region,
                                          new URLCallback() {
        public void urlArrived(URL u) {
            QueryResponse r = new URLRedirectQueryResponse(u);
            cb.responseArrived(r);
        }
    });
}
```

CSP-based alternatives, such as Erlang-based systems take concurrency to the other extreme: Behavioral elements execute as "processes" that communicate over asynchronous channels. At massive scale, for maximum efficiency and thus lowered hosting costs, the more esoteric approaches of callbacks or CSP would merit serious consideration.

2.5 The Bridge Builders

With this bridge-based top-level design, we have in effect created a third developer role. This is an interesting role: It calls for a person or team who is familiar with networking issues, and who knows a little bit about developing code for servers and code that runs on client devices. It is tempting to pigeon-hole developers along software packaging boundaries; often, there's "the client team" and "the server team." Server programming typically requires

an in-depth understanding of concurrency, packaging and deployment strategies, scalability, reliability and other topics. Client-side programming typically requires an in-depth understanding of the consumer electronics target devices, a very different but equally complex mechanism for software packaging and provisioning, a different kind of concurrency, dealing with platform fragmentation, development in a resource-constrained environment, etc. The role of bridge programmer, however, requires *neither* in-depth server programming knowledge nor in-depth device programming knowledge. This module can be developed by a team with in-depth knowledge of networking and protocols, and an ability to program relatively simple "bridgehead" adapters on both the server and the client side.

It is important to consider the perspective that a bridge builder would have. We expect that the perspective of the bridge team would most likely align with that of the client software team. That is, the bridge team's focus would be on the responsiveness and reliability that goes into the user experience of the end customer. The output of the bridge module is on the client device, and in terms that map closely to the user experience. As a consequence, a bridge programmer would be directly exposed to the customer's perspective of the system. By way of contrast, developers on back-end systems tend to place a greater emphasis on cost, efficiency, throughput, scalability, and other provider-centric measures. This is as it should be, and speaks to the value of clarity around development roles.

In section 2.4, we showed what the client portion of the bridge code would look like. The style of the bridge code is impacted by the server architecture (e.g. callback-based vs. traditional in-line code), but beyond that, the bridge module's implementation need not be overly concerned with the more in-depth issues of server development. Naturally, the inner details of RPC mechanisms, the database scheme, tuning for concurrency, caching policy, heap management, state management, etc. continue to be vitally important. These issues, however, can be encapsulated behind the in-process API that is offered to the bridge module from the back-end modules that make up the system. This leaves the bridge team free to negotiate robust, stable APIs on both the server and client sides, and to implement a networking protocol that leads to maximum efficiency and responsiveness.

3 GRIN Scene Graph

An example of a system that breaks the linkage between software packaging boundaries and system architecture is the GRIN scene graph [6]. GRIN is a scene graph framework for presenting interactive graphics with high production values, targeted at devices connected to a television receiver, like Blu-ray. One interesting aspect of this environment is that some players do not have hardware floating point units, so floating point calculations can be two orders of magnitude slower than integer calculations. This can cause difficulty when one wishes to make fluid user experiences featuring motion that mirrors real-world physics with sufficient fidelity.

One way around this difficulty is to pre-compute animation paths. This is functionality that was added to GRIN, in a way that makes it transparent to the application programmer whether the computation of the path is done at runtime or at build-time. An animation path is described declaratively, as in the following example:

```
0       255     start
11        0     ease-in-out-quad
23      255     ease-in-out-quad
```

This says "At frame 0 start at position 255, animate to position 0 using a quadratic ease in/out function until frame 11, and then animate back to position 255 using a quadratic ease in/out function until frame 23." In this environment, animation is typically done at a fixed frame rate, so "frame" is an appropriate unit of time, but some other time unit could have been used instead.

From the above, it's not clear if the easing function is executed at build time or runtime. When targeting Blu-ray devices, the GRIN framework pre-calculates fixed points along the path, calculating sufficient interpolation points such that runtime integer linear interpolation along the path is constrained to be within an acceptable error bounds. If the same scene graph were targeted to an environment where floating point performance is assured and where data transmission costs are greater, this could be implemented for that target as a runtime computation. Transparently moving computation in this way was indeed envisioned for a future extension of GRIN.

In this system, the GRIN framework serves as, in essence, a bridge between the build-time environment and the runtime environment. It was designed to allow computation to move from one unit of software packaging (the GRIN scene graph compiler) to another (the GRIN runtime), without exposing the application developer to the difference.

An operative concept in this scheme is the separation of what the application is to do from how it does it. Moving toward declarative application formats where appropriate can help. An equivalent animation would typically be done in a web application with procedural JavaScript code doing floating-point computations directly. There would be numerous practical difficulties in attempting to pre-compute the formulas at build time, even in those cases where doing so is theoretically possible. Further, the distinction between when it is possible and when it is not would seem arbitrary and unpredictable to the script writer.

Declarative formats, by themselves, are no panacea either. If a purely declarative format were adequate for the production of modern user interfaces, then "plain" HTML without scripting would have won the day. Declarative formats and some level of client-side programming exist in synergy. The declarative animation paths of GRIN move one area of animation from the procedural to the declarative, in a way that does not rely on limiting the author to a set of "canned" formulas: The set of animation path formulas is extensible, with the same linear interpolation runtime on devices with poor floating point support, and direct evaluation on devices with good floating point support. The decision of where to do the computation is

contained entirely within the GRIN module; client code does not change as a result. In this way, GRIN's scene graph encapsulates this design decision, and achieves lower coupling in the resulting system than would otherwise be the case.

Compiling a scene graph that contains, among other things, an animation path is a different kind of bridge than the client/server examples given earlier. However, it centers around the same notion: In today's heterogeneous systems, moving functionality from one physical device to another, or from one point in time to another can offer substantial benefits. If the software architecture is built along the lines of the packaging of these implementation artifacts, it impedes the evolution of the system, due to the tight coupling between modules that results from the large surface area of the interfaces between them. Conceptualizing such bridges as first-class citizens in the software architecture is an effective tool in managing and containing this aspect of system complexity.

The GRIN application of the bridge idea might seem less controversial than the client/server scenario presented above. After all, computer language compilers often do optimizations that trade off space vs. runtime speed; the GRIN scene graph compiler is no different conceptually. This provides validation for the value of separating system architecture from software packaging boundaries. Aside from the performance impact, the application developer's modules are unaffected by the packaging decision.

4 Module, Defined

After having explored different modular decompositions, we come back to the proposed definition of a module from the introduction:

> A module is a collection of one or more software artifacts into a cohesive whole that has defined behavior and defined interfaces to other modules.

The essence of a module is that it is something that has a definition. A module is a mental construct that is (hopefully) expressed in that definition. When realized, a module includes one or more software artifacts, but the essence of the module is its definition. The definition of a module specifies its behavior and its interfaces to other modules.

Picking the boundaries of a module is a way of organizing a system, so that the people who create, use and maintain it can understand it better. An analogy can be drawn to a bookstore. Many different organizations of a bookstore are possible, and they all work, but some are more effective than others. For example, a bookstore where the books are arranged by publisher, and by author within a publisher might be convenient for re-stocking, but that organization is not very effective for customers who come into the store looking for a book on Gardening. In a way, this is like organizing a software system by software packaging boundaries, rather than by conceptually coherent areas of functionality. How a bookstore - or a software system - is

organized is of less importance to someone who has years of experience with it, and knows where everything is to be found, down to the minute details. The importance of effective organization is most apparent when a system needs to change, when the people working on it change, or when someone new to the system is brought in. It is often hard for the creator of a complex system to fully appreciate the impact of a different modular decomposition.

A finished software module usually includes one or more collections of code. In the client/server example given above, the module consists of the two "bridgehead" collections of classes, the specification of their interfaces, the build rules, any module-level tests, and of course the specification of the module's behavior and interfaces. Module-level tests will ideally exist, and have minimal dependencies on other modules. This can be done by providing "mocks" of the module's collaborators. In our client/server example, this could be a small amount of test scaffolding that includes hard-coded data values within an implementation of the back-end APIs, and some kind of test script to drive the client side of the APIs. The client side might be compiled against a platform definition that is limited to the client device, but it might be executed in a simulation environment (e.g. as a desktop Java program), with perhaps a `PipedInputStream` / `PipedOutputStream` pair connecting the client bridgehead to the server bridgehead within the same process.

A module might include other tools, such as emulators or compilers. In the GRIN example, the GRIN scene graph module can be though of as including the runtime GRIN library that executes on the client device, the scene graph compiler, the build rules, and the "GrinView" emulator. This emulator is a desktop Java program that allows a developer to run and debug their scene graph on a development machine, rather than running it on a Blu-ray player where the debugging facilities are limited, and the turn-around time is slow. The emulator can be considered to include the build rules for the emulator; these build rules include support for a design pattern for stubbing out areas of application functionality that are too dependent on player functionality to be emulated in the current desktop environment.

As the above examples show, the physical manifestation of a module differs substantially, depending on the environment and the problem being solved. A module is, before anything, a mental model used by people to understand a system.

5 Conway's Law, Brooks' Law and The Tower Of Babel

> *Come, let us go down, and there make such a babble of their language that they will not understand one another's speech. [7]*

Conway's Law, Brooks' Law and Parnas' criteria for what makes a good module are related in interesting ways. Conway's law, published in [8], states "... organizations which design systems ... are constrained to produce designs which are copies of the communication structures of these organizations." [9] Brooks' law is a distillation of [10]. It's often summarized

as “adding manpower to a late software project makes it later,” “ or even “nine women can’t make a baby in one month,” [11], but more generally, Brooks’ thesis is that adding people to a project increases the communications overhead, and beyond a certain point, adding people decreases the amount that the group gets done.

Nikolai Bezroukov explores the relationship between Brooks’ Law and Conway’s Law in [12]. He observes that the essence of Brooks’ law is deeper than its typical formulation. Brooks “postulates [a] complex relationship between [the] structure of the project and [the] structure of the team of developers involved.” One reason why making a team larger doesn’t necessarily increase the team’s capacity has to do with how well that new member fits in with the team structure, which, as Conway points out, is influenced by and influences the system architecture. Viewed in this way, the present paper is not simply arguing that a critical bridge component should be conceptualized as a top-level module. We are arguing that ownership of this bridge module should be a distinct role within the development organization.

In essence, Parnas provides a tool that helps to ameliorate the effect of Brooks’ law. An effective modular decomposition that achieves low coupling and high cohesion allows sub teams greater autonomy, thus reducing the communications burden that is at the heart of Brooks’ law. Conway’s law expresses the linkage between the modular decomposition and the team structure.

As software systems have become larger and more complex over time, developer roles have become more specialized. Each specialization develops its own vocabulary, sets of tools, accepted best practices, etc. The interaction between e.g. Android client developers and server developers can come to resemble The Tower of Babel, with different jargons and different frames of reference resulting in poor communication, frustration, errors, and lost opportunities. The bridge module fulfills the technical role of converting an interface that makes sense on one side of the bridge (typically an in-process API) into an interface that makes sense on the other (typically an in-process API running within a very different platform). The developer role associated with the bridge module fulfills a similar role in facilitating smooth and productive communication among the teams. This “bridge team” need not know all of the intricacies of the technology platform at either side of the bridge. They need to know enough about both sides to speak the right language, and to convert between the two. As noted in section 2.5, this bridge team is likely to have a perspective aligned with the results-producing side of the bridge, which typically makes the developers sympathetic to the concerns of the end customer. The role of bridge builder is important, and should not be neglected, nor should it be pushed up to management to resolve. Better, faster and more maintainable results can be achieved by making this role explicit, both in terms of developer team roles, and in the top-level modular decomposition of the system that, inevitably, reflects that team structure.

6 Conclusion

Modern heterogeneous software environments have made the design space for computer systems more complex. The field of computing has developed many tools and techniques to build these complex systems. Despite this, we still experience brittle, expensive systems that are hard to maintain.

Parnas' insights on decomposing a system into modules were highly successful in introducing modular programming, and in influencing the nature of object oriented design and object oriented programming. These insights can be applied to heterogeneous systems. By maximizing information hiding, where "information" is defined broadly, we can achieve modules with low coupling and high cohesion. This leads to simpler, more "beautiful" designs whose realizations are more reliable, more maintainable, and faster.

Much as in the time of Parnas' paper, this way of decomposing systems into modules goes against the way many systems are built. His paper was initially rejected for publication. One reviewer stated, "Obviously Parnas does not know what he is talking about because nobody does it that way." [3] The idea of breaking a system architecture along the boundaries imposed by network topology or by artifacts of the packaging of software executables is intuitively appealing. This way of doing things is reinforced by a tendency to apply superficial criteria to developer roles, like exposure to a particular programming language, rather than understanding the kind of system development done in a given role. Dealing with a certain kind of concurrent programming is much harder to learn than a programming language, but all too often developer roles are pigeon-holed by superficial criteria, such as "knows Java" or "Pythonic." Since particular languages tend to go with particular packagings of software executables, this reinforces the architectural choice of dividing modules along packaging boundaries, in line with Conway's Law. This design decision results in tight coupling between modules, and modules that are sensitive to arbitrary decisions about where computation is to be performed that are typically made early in the development cycle, before performance is well understood, and before the inevitable changes in the features of the end product. Expensive, brittle systems are the result.

Superior results can be achieved by applying Parnas' criteria to modular decomposition, without undue regard to packaging boundaries. In many cases in a distributed system, the protocol between the client and server is precisely the kind of arbitrary design decision that is subject to change as the system is enhanced and optimized. As such, it is appropriate to encapsulate this within the boundaries of a module, where changes can be managed without disrupting more of the system. This can be thought of as a bridge module, where the bridge includes its land-based connections on either side. An analogous bridge is pictured in figure 10 on the next page. A software bridge module consists of a communications protocol, and the adapter code on each side of the bridge that translates into the domain of its peer modules on either side. This adapter code can be created by someone with a general familiarity with the implementation technology; it usually does not require deep familiarity with the platforms in

Figure 10: A bridge that includes the connections on both sides within the module boundary.

question. The role that is called for is one of translator, and the module that is needed is one that provides the connection between two domains.

Another application of Parnas' criteria is to encapsulate the decision of when to carry out computation within a module. This can even result in different implementations being used in an automated fashion, depending on the deployment target. By effectively encapsulating this decision, the rest of the system can be insulated from the complexities inherent in e.g. moving computation from run time to build time.

Parnas' ideas were at the core of modular programming, and provided much of the basis for object-oriented programming. In today's heterogeneous systems, the key notion of information hiding can be applied in a new way, resulting in faster, less expensive, more reliable and more maintainable systems.

References

[1] Parnas, David. On the Criteria to Be Used in Decomposing Systems into Modules. *Comm. ACM, Vol. 15 (12),* 1972, pp. 1053-1058.

[2] Rogers, Wm. Paul. Encapsulation is not information hiding. *Java World,* May 18 2001, `http://javaworld.com/javaworld/jw-05-2001/jw-0518-encapsulation.html`, retrieved May 15, 2012

[3] Parnas, David. The Secret History of Information Hiding. *Software Pioneers,* ISBN 3-540-43081-4, pp. 399-408.

[4] Foote, William et al. Systems Architecture is not Network Topology: Connecting the Consumer Device. JavaOne 2010 session.

[5] Communicating Sequential Processes, Wikipedia, `http://en.wikipedia.org/wiki/Communicating_sequential_processes`, retrieved June 4, 2012

[6] HD Cookbook - GRIN Scene Graph, `http://jovial.com/hdcookbook/grin.html`, retrieved April 11, 2012

[7] Genesis 11:1-8, quoted in Brooks, Frederick P. Jr., *the mythical man-month,* ISBN 0-201-00650-2, p. 73.

[8] Conway, Melvin E. (April, 1968), "How do Committees Invent?", Datamation 14 (5): 28–31

[9] Conway's Law, Wikipedia, `http://en.wikipedia.org/wiki/Conway's_law` retrieved April 11, 2012

[10] Brooks, Frederick P. Jr., *the mythical man-month,* ISBN 0-201-00650-2, pp. 13-26.

[11] Brooks' Law, Wikipedia, `http://en.wikipedia.org/wiki/Brooks'_law` retrieved April 11, 2012

[12] A Note on the Relationship of Brooks Law and Conway['s] Law, Version 0.1, `http://www.softpanorama.org/Articles/note_on_the_relationship_of_brooks_law_and_conway_law.shtml` retrieved April 11, 2012.