Dynamic On-line Object Update in the Grumps System

Huw Evans
Department of Computing Science
Glasgow University
Glasgow, Scotland

huw@dcs.gla.ac.uk

Abstract

This position paper describes the Grumps approach to the on-line upgrading of objects in a distributed system. This is supported by three main, well-known ideas: core code abstractions are defined in terms of Java interfaces and not classes; as many decisions as possible are delayed until run-time; and object containers are used to manage groups of objects at run-time. The advantage of the first idea is that it introduces a level of indirection in between what an object does (in terms of its interface) and how that is accomplished (in terms of its class). The advantage of the second idea is that it does not force the programmer to make decisions at compile-time that may be hard to change later; they can change their mind and easily alter the system's behaviour at run-time. The use of the third idea allows objects to be grouped together and the container can act as a representative for or abstraction over the contained objects. The major Grumps abstractions are described, an example object upgrade is presented and the advantages and disadvantages of this approach are discussed.

1. Introduction

The Grumps[1] project [1] is developing techniques and software to collect, manage and analyse large collections of user actions automatically. The project has four interrelated goals: to make it easier to organise investigations that efficiently and as unobtrusively as possible collect and store traces of remote users' actions; to make it easier to analyse the stored data to test ideas about the users' activities and the facilities provided to support them; to discover whether such an approach is effective in improving the quality of distributed information systems; and to trial these issues in application areas such as education and bioinformatics.

The requirements for the distributed execution architecture are that: it can scale to the level of the Internet; it is able to work via firewalls, proxy servers, systems using network address translators (NATs) and dynamically-assigned IP addresses; the topology can be changed at run-time; and all deployed objects (and their implementations) can be updated during system execution. The first two requirements are described in [6]. This paper focuses on the third requirement.

The rest of this paper is organized as follows: section 3 introduces the main Grumps abstractions; section 4 shows how object-upgrade is performed; section 5 discusses the pros and cons of this approach; section 6 briefly outlines the constraints on evolution following this approach; section 7 discusses Grumps evolution in a distributed system; section 8 briefly details how code can be ported to the Grumps system and section 9 summarises the paper.

[1]This paper discusses a snapshot of work in progress. See `http://grumps.dcs.gla.ac.uk/` for up-to-date information.

2. Motivation

This section describes why the Grumps approach has chosen to work from scratch, rather than base the implementation around some other technology, e.g., Enterprise JavaBeans [3], or some standard Java technology, such as classloaders.

2.1. Support for Evolution in Java

Java is an object-oriented language that is statically compiled from source code into bytecode. This bytecode is then interpreted by a virtual machine which may, optionally, compile the bytecode into native machine code to increase application performance. The Java language is quite rigid in its support for dynamic on-line object update and, in particular, no part of the language *explicitly* addresses the problem. The standard object-oriented concepts of encapsulation, polymorphism and dynamic dispatch can help support on-line object update, but they are not directly aimed at providing an object update protocol.

Part of the problem is that the Java language does not define a powerful reflection mechanism. The standard Java reflective capabilities allow a programmer to discover information representing the current state of a program, but they do not allow them to interact with the ongoing computation in a more powerful way, e.g., by dynamically changing the meaning of a method call. To address this problem, other researchers have created Java meta-object protocols that work by manipulating the bytecode, e.g., [8]. Others have worked on support within the virtual machine to allow for more exploratory forms of Java development, e.g., [2]. However, these solutions are not part of the standard Java language.

Some decisions in Java virtual machines are hard-wired into the virtual machine's technology. For example, when a class can be unloaded is not something that a programmer has any access to. Typically, this is something that the virtual machine does to free up memory without informing the programmer via an API. A Java programmer can gain more power at run-time by using their own classloader, although they cannot force a class to be unloaded. However, using a classloader can lead to other problems as Java defines run-time type equivalence in terms of the fully-qualified name of the class plus the identity of the classloader that loaded the class. Classloaders are useful, e.g., Java applets use them to load classes from a web site. However, writing a classloader can be difficult as the programmer has to be careful to ensure classes loaded by one classloader are not used in the context of classes loaded by a different classloader. If they are, `java.lang.ClassCastException`s can be unexpectedly thrown. Also, in a system where classes are being updated at run-time, for the type system to consider a class to be the same as a previous version, the instance of the classloader that loaded it must be retained and the instance much be capable of reloading the new class.

This situation is made worse because a Java library may make use of its own classloader. If a reference to a classloader is not exposed (which typically it is not) and as Java does not define a meta-level, it is not possible to affect how this classloader behaves. This may not be a problem for some applications where the classes used by one library may not require evolution. However, in the general case, using classloaders in Java to provide a form of class evolution is not possible.

2.2. Enterprise JavaBeans

The Enterprise JavaBeans (EJB) 2.0 specification [3] describes a component architecture for the development and deployment of component-based distributed business applications. This version of the specification has integrated the Java Message Service and has introduced an asynchronous, message-driven bean invocation model as well as defining a declarative syntax for defining query methods that can be used to find persistent objects. In addition, improvements have been made to the persistence model and the use of home and local interfaces. Enterprise JavaBeans were not used in the development of Grumps because it was not clear how effective JavaBeans would be at supporting the model described in the rest of this paper. By choosing to use EJB, one particular event model would be forced upon the programmer. Also, EJB requires the programmer to use Java's Remote Method Invocation system (RMI) for remote communication. One requirement for Grumps is to be able work with the technology that is routinely deployed onto the public Internet, e.g., firewalls and dynamically-assigned IP addresses. RMI does not negotiate firewalls very well; having to use it as an underlying communication technology would make programming the more Internet-related aspects of the system quite difficult.

In addition to the above, by building a system from scratch, greater control over the mechanisms for updating a system at run-time can be retained. This is because no other mechanisms have to be used, e.g., EJB, and worked around should they not be amenable to update during system execution.

3. Core Grumps Abstractions

The communication support system in Grumps is referred to as a GrumpsNet. A GrumpsNet consists of a graph of communicating objects, executing over a number of computing devices (e.g., host computers, PDAs) which are connected together via a (possibly wireless) network. Each device may be running a number of Java virtual machines (JVMs) and each JVM will contain a part of the graph of deployed objects.

In Grumps, objects communicate with each other in terms of events which are sent through channels. Two main kinds of object are defined, GrumpsContainers and Grumps Units, and three kinds of channel. These abstractions and how they are used are described below.

3.1 Input and Output Channels

Grumps events are communicated via input and output channels. An input channel is defined to be the receiving end of a communication line, which can be read to receive events. An output channel is the sending side of the communication line. Events that are written at the sending side are read at the receiving side and one output channel is connected to exactly one input channel. The sending side and the receiving side can be either in the same process or in different processes that are running on different machines. Event channels are implemented using sockets, so the usual amount of transient buffering is available to the Grumps programmer. Grumps additionally defines a more reliable kind of buffering in the form of a persistent event store-and-forward object. In the current Grumps implementation, this object writes events to disk and periodically checks to see if it can send the persistent objects to a target object for eventual processing. In this way, Grumps provides some support for disconnected event collection.

The distributed fault model for Grumps is the same as that provided by the underlying communication facility of sockets. Currently, Grumps does not define any other semantics over and above those provided by the Java socket implementation. This is straightforward and easy for the user to understand. However, the consequences are that a different number of distributed communication error kinds are exposed to the programmer, some of which could be abstracted over, e.g., at least once message delivery.

3.2 Grumps Containers, Units and Events

A GrumpsContainer is an object that encapsulates a number of Grumps Unit (GU) objects (figure 1). All Grumps objects communicate with each other in terms of events that are implemented as Java objects. GrumpsContainer and GU objects each define a single control event-channel to which event objects can be sent. The event object then operates over the target GrumpsContainer or GU object.

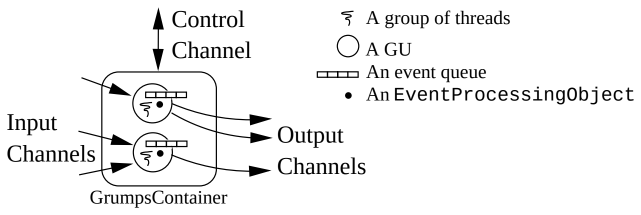

Figure 1. A Grumps Container

Control events are sent to this channel synchronously[2] and this class of events carry Java code with them. Each control event defines a method called `apply` that takes an instance of GrumpsContainer or GU as a parameter and which contains code to perform the actions of the event. When a control event arrives at the GrumpsContainer[3], a reference to the GrumpsContainer is passed to this method. The `apply` method then executes, calling methods on the GrumpsContainer. In this way, a GrumpsContainer responds to an open-ended, evolvable set of events. This is preferable to hard-wiring into the GrumpsContainer the types of the event that it can respond to. A programmer can create a new control event, supplied with new code, and send it to a GrumpsContainer which will run the code on itself.

Each GU object has a number of input and output channels and an object (called the EventProcessingObject) that processes the events the GU object receives. Input and output channels communicate in terms of Grumps Events which are sent asynchronously.

A Grumps event carries with it information on when the event occurred, which object originally sent it and some investigation-specific information. Each event arriving on an input channel is placed into a FIFO queue, one queue per channel. This queue is managed by the EventProcessingObject, which is responsible for reading the objects from the queue, processing them in some way and (possibly) sending them out on an output channel. GUs are typically combined into graphs of GUs and it is the intention of the Grumps project to be able to treat a particular local graph of GUs as a single GU. This allows users of the system to reuse components, in order to build sophisticated investigations from a collection of GUs with well-known behaviour.

[2]Communication is synchronous to the control channel as it is generally useful to be able to get a result object back as a result of sending in a control event.

[3]The actions performed at a GU are similar and are not discussed.

4. Installing and Updating Objects at Run-time

When a programmer starts a Grumps JVM, it only contains the Grumps run-time system. The programmer then populates it with GrumpsContainer and GU objects which are given their EventProcessingObjects at run-time. Each GU object is given a name by the programmer. This allows the system to be able to distinguish between different GU objects held in the same GrumpsContainer. When a GU object needs to be updated, e.g., its EventProcessingObject is to be replaced, a reference to the GU in question is first of all retrieved by passing the name of the GU to the containing GrumpsContainer. Section 4.1 discusses how a new GU object is installed into a pre-existing GrumpsContainer object and section 4.2 then discusses how the EventProcessingObject object in this newly installed GU object can be replaced.

4.1 Installing a New GU

In order to install a new GU in a GrumpsContainer, the program has to first of all gain a reference to the GrumpsContainer's control channel. A ConnectionRequestControlEvent is then created. This object is provided by the Grumps architecture and it handles installing the new GU object (with its EventProcessingObject) in the GrumpsContainer. When the ConnectionRequestControlEvent arrives at the remote GrumpsContainer, a reference to the GrumpsContainer is passed to the event's `apply` method. The `apply` method then creates a new GU object, initialising it with the EventProcessingObject carried in the control event. This GU is added to the GrumpsContainer and the threads for the GU and the EventProcessingObject are started (so that they can process incoming events). The new GU has a name associated with it. The name object contains a channel so that other objects may communicate with the GU. This name object is passed back to the program that sent the

ConnectionRequestControlEvent so that it can then communicate with the newly installed GU.

4.2 Performing an On-line Update

In order to perform an on-line object update of an EventProcessingObject within a GU object, the programmer gains a reference to the control-channel of the GrumpsContainer that contains the GU of interest. An `UpdateEPOControlEvent` is created which carries with it the name of the GU that is to have its EventProcessingObject changed and the new EventProcessingObject instance. The GrumpsContainer interface defines a method `getConnection` which takes the name object as a parameter. This method returns a reference to the GU object which has its `setProcessingObject` method called to replace the current EventProcessingObject with the new one. As we have installed a new EventProcessingObject object, its thread has to be started and `null` is written back to the process that sent this control event to inform it that the object has been successfully updated. Currently, the programmer has to deal with the case of managing the shutting-down of the currently executing EventProcessingObject and starting the thread for the new EventProcessingObject. In addition, no way of indicating the failure to install the new EventProcessingObject is provided. Dealing with these cases is an area for future work.

5. Discussion and Related Work

As GrumpsContainers and GU objects respond to an open-ended, evolvable set of events that carry their code with them, decisions about how to process a deployed object can be made at run-time. If the programmer (or a user, via a tool) wants to process a deployed object in a different way, a new event is written that encapsulates the new code. This event is sent to the deployed object and the new code is executed over that object.

However, this level of flexibility does leave the architecture open to attack. Currently, no checks are performed on the events that are received. It would be possible to add security features to an event, and, although this is not a major focus for the Grumps project, some support may be provided in the future.

In a distributed system, it is possible for a deployed GU object to be updated by an event before new event code can operate over that GU. This kind of issue has to be taken care of at run-time by ensuring that the ordering of events arriving at a GU will cause its state to be changed in a way that is compatible with the rest of the distributed system. This can be hard to code and future versions of Grumps should provide tool support to address this issue.

In the Grumps implementation the majority[4] of the abstractions are expressed using interfaces and not classes. This makes the support for object and implementation update as simple as object assignment. After the update has been performed, the old object may be garbage collected and, in certain circumstances, the class of the replaced object may be unloaded by the virtual machine. After this update, if another new implementation is required, a new object can be assigned, that implements the same (or an extended) interface, but does so with a different class.

An alternative implementation technique would be to build the core Grumps abstractions in terms of classes. However, in this case, we can only provide new implementation updates in terms of a sub-class of the class specified in the code. This means that implementation update can only be defined in terms of extensions to pre-existing super-types that must be retained in the run-time VM. It is possible to address some of these problems by using a Java classloader, however, this brings its own problems; Java defines type equivalence in terms of the fully-qualified class name and the identity of the classloader that loaded that object's class. In a system that needs to support implementation replacement, this can be too inflexible as the classloader instance has to be retained to ensure type-compatibility.

However, the flexibility of the object-assignment approach does come at a price: if any state needs to be copied between the old and new objects, the programmer has to write code to manage this; the programmer has to be careful about updating objects that use threads and synchronization; and operations may have to be performed on the old object before it can be safely replaced. Providing solutions for these problems within the context of an object-upgrade protocol is an area for future work.

Software architectures for distributed systems (e.g., C2 and the ArchStudio framework [7]) are related to the Grumps work described here. These kinds of architectures are typically built from components which are connected together using pipes down which messages travel. Such systems are integrated together through guidance from a human, using a composition language or a GUI tool. Although this approach allows some degree of change at run-time (e.g., components can be removed and pipes reconnected), these systems tend to only allow complex objects, such as components, to be replaced. The Grumps project is focussed at the individual object-level as the basic unit of replacement.

Tool support for the programmer is an area for future work as Grumps programmers will need to be able to visualize the running system and express to it how objects should be replaced.

[4]GrumpsContainer is currently a class. This shall be made an interface in a later version of the Grumps software.

6. Constraints on Evolution

It is currently assumed that the interfaces for the GrumpsContainers and the GUs do not change. However, in future work it is hoped the container model will support the ability to change interfaces at run-time. In the current system, a programmer may use the GU type within a GrumpsContainer. This is expressed in the code as methods on the GrumpsContainer interface take GU types as parameters and pass them back as results. Thus, the GrumpsContainer may be seen as a scoping mechanism for the use of the GU type. If an updated GU type had to be used, that could not be represented as a subtype of the original GU, a new kind of GrumpsContainer could be provided that made use of it. In this way, a number of different versions of GrumpsContainer and GU would be defined, with the outer instance acting as a boundary for changes to the type of the inner instance. This approach could be generalised and applied to the GrumpsContainer and the object that contains it. Pursuing this possibility is an area of future work currently being considered as part of the object-update protocol.

7. Evolution in a Distributed System

So far, the discussion of on-line object update has only involved changes to one instance. In a distributed system, several objects that are in different address spaces may need to be updated concurrently and consistently. Although support for this kind of evolution is an area for future work in Grumps we briefly outline our current thinking on this topic.

In order to perform the kind of investigation briefly discussed in the Introduction, it is assumed that the user will first of all design their investigation using the Grumps approach, its software and tools. In order to manage this activity, the Grumps team intends to provide a database in which the user's current view of the investigation will be stored. This database and its associated tools will be used to design the investigation, to deploy it and to manipulate it at run-time. To support evolution in a distributed system, the database will contain a data repository and an analysis repository. The data repository will contain the data model and schema for the current investigation. This will contain information on the network of data-collection and processing entities that should be deployed to perform the investigation, together with details of how they should be inter-connected. This network will be generated by processing stored source-code repositories within the context of the data model and schema. The analysis repository will allow the investigator to store and perform queries over the current (possibly executing) investigation. This will allow them to, amongst other things, query the run-time system to see if answers that help them meet their investigation goal can be found. In order to update objects in the distributed system in a consistent way, the investigator will first of all query the database and running system. The answers to these queries will be then be used by the investigators to update objects in the distributed system via the database and the Grumps tools. It is assumed that the tools will use the information in the database and the running system in order to perform the updates to the distributed system using some form of a distributed transaction. System downtime will be minimised and the work on DRASTIC [4, 5] will inform the solution for updating the system in the face of currently executing events.

One important issue that is still outstanding is how to track which measurements were collected before an evolution and which were collected after an evolution. This is an area for future work.

8. Porting Code to Grumps

In order to port non-Grumps code to the Grumps system a programmer has to first of all place their application-level code in a class that implements the GU interface. In order for the application-level code to communicate with other objects, Grumps communication primitives should be used. This is to ensure that any object-upgrade technology provided in the future will be accessible to this code. The programmer must, therefore, identify a set of GrumpsEvents that their code responds to and a set of events that it will generate. Once this GU implementing class has been defined, the class has to be instantiated and installed into a GrumpsContainer as described in section 4. From this point on, the code has been ported to the Grumps system and it will act as any other GrumpsContainer and GU instance.

9. Summary

The approach to on-line object upgrade in the Java-based Grumps system has been described. The contribution of this paper is the demonstration that by combining three well-known ideas — that of building abstractions in terms of interfaces, delaying as many decisions as possible until run-time, and containing groups of objects inside another — a system can be implemented in which run-time object upgrade can be more easily performed. There are two areas of future work to be considered: dealing with malicious events and defining an object-upgrade protocol.

10 Acknowledgements

The author would like to thank the other members of the Grumps team for providing the necessary environment in which to perform this work and the UK EPSRC (GR/N38114) for providing the funding.

References

[1] Malcolm Atkinson, Margaret Brown, Julie Cargill, Murray Crease, Steve Draper, Huw Evans, Philip Gray, Christopher Mitchell, Martin Ritchie, and Richard Thomas. Summer Anthology, 2001. Technical Report TR-2001-96, Department of Computing Science, Glasgow University, September 2001.

[2] L. A. Chamberland, S. F. Lymer, and A. G. Ryman. IBM VisualAge for Java. *IBM Systems Journal*, 37(3):386–408, 1998.

[3] Linda G. DeMichiel, L. Umit Yalcinalp, and Sanjeev Krishnan. *Enterprise JavaBeans Specification, Version 2*. Sun Microsystems, 2550 Garcia Avenue, Mountain View, CA, 94043, v2.0 edition, August 2001.

[4] Huw Evans and Peter Dickman. DRASTIC: A run-time architecture for evolving, distributed, persistent systems. In Mehmet Aksit and Satoshi Matsuoka, editors, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP '97)*, volume 1241 of *LNCS*, pages 243–275, Jyväskylä, Finland, June 1997. Springer.

[5] Huw Evans and Peter Dickman. Zones, Contracts and Absorbing Change: An Approach to Software Evolution. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA '99)*, volume 34 of *SIGPLAN Notices*, pages 415–434, Denver, Colorado, USA, October 1999. ACM.

[6] Huw Evans, Peter Dickman, and Malcolm Atkinson. The GRUMPS Architecture: Run-time Evolution in a Large Scale Distributed System. In *Workshop on Engineering Complex Object-Oriented Solutions for Evolution (ECOOSE)*, colocated with OOPSLA, Tampa, Florida, USA, October 2001.

[7] Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. Architecture-based runtime software evolution. Technical Report ICS-TR-97-38, University of California, Irvine, Department of Information and Computer Science, September 1997.

[8] Ian Welch and Robert J. Stroud. Kava-Using byte code rewriting to add behavioural reflection to java. In *Proceedings of the 6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS-01)*, pages 119–130, Berkeley, California, January 29– February 2 2001. USENIX Association.