Safely Updating Running Software

Tranquility at the object level

Peter Ebraert[*]
Universiteit Antwerpen
Middelheimlaan 1
Antwerpen, Belgium
peter@ebraert.be

Hans Schippers
Universiteit Antwerpen
Middelheimlaan 1
Antwerpen, Belgium
hans.schippers@ua.ac.be

Tim Molderez
Universiteit Antwerpen
Middelheimlaan 1
Antwerpen, Belgium
Tim.Molderez@ua.ac.be

Dirk Janssens
Universiteit Antwerpen
Middelheimlaan 1
Antwerpen, Belgium
dirk.janssens@ua.ac.be

ABSTRACT

Software applications are updated frequently during their life cycle. In order to do so, they usually must be shut down, adapted and restarted. This causes periods of unavailability, which is not acceptable for some applications. Dynamic software updates (DSU) – in which applications are updated at runtime – is a technique that can be used to update software without the need to restart it.

One of the problems of DSU is ensuring state consistency of the active application. Tranquility has been proposed as a necessary and sufficient condition for ensuring state consistency at the granularity of software components. As many object-oriented applications do not have a notion of software components, we aim to introduce tranquility at object granularity.

1. INTRODUCTION

An intrinsic property of a successful software application is its need to evolve. In order to keep an existing application up to date, we continuously need to adapt it. Usually, evolving such an application requires it to be shut down, because updating it at runtime is generally not possible. In some cases, this is not acceptable. The unavailability of critical systems, such as web services, telecommunication switches, banking systems, etc. could have unacceptable consequences, both for the software users and the producers. Additionally, shutting down a running application strongly hinders self-adaptation.

[*]Dr. Ebraert is funded by the "Agentschap voor Innovatie door Wetenschap en Technologie" via the Optimma research project.

A solution for this problem lies in the domain of Dynamic Software Updates (DSU), in which a part of the system is updated while the system itself remains active. In order to do so, the system must reside in a consistent state before the change is performed [11]. A consistent state is a state from which the system will be able to terminate correctly.

Kramer and Magee define a system as a directed graph whose *nodes* are system entities and whose *arcs* are connections between those entities. Nodes can only affect each others states via *transactions*, which consist of a sequence of messages that must be executed atomically (i.e. either all messages are executed, or none of them are). During their execution, however, the state of these transactions is distributed in the system and may temporarily leave different nodes in a mutually inconsistent state. Therefore, nodes can not always be updated during the execution of a transaction without breaking application consistency. The notion of tranquility was introduced in [21]; it is used to indicate when a software component [18] is in a consistent state, at which point it can be updated.

In the context of DSU, the use of software components yields two important advantages. First, a component is a black box, which guarantees that other components do not rely on its implementation details. Second, for each service a component requires, it is known which other components participate in the transaction. This information can be used in order to determine which components should be in tranquil state for a dynamic software update to be allowed to proceed.

In practice, however, most systems are not designed with a component framework in mind, hence the tranquility approach is not applicable.

This paper adapts tranquility such that it can also be used in the absence of a component framework. More specifically, we will apply it in a classical class-based object-oriented scenario, without relying on the properties of component-based systems. The remainder of this paper is organized as follows: Sec. 2 revisits the definition of tranquility as it was introduced in a component-based environment. Sec. 3 discusses the relevant similarities and differences between components and objects. Sec. 4 continues with an example scenario demonstrating the use of tranquility. An overview of the implementation of tranquility is then presented in Sec. 5. The paper is concluded in Sec. 7.

2. TRANQUILITY

Tranquility starts from the observation that a good approach for enhancing reusability and decoupling the system parts, lies in a black-box design of system nodes. This implies that nodes may require services from other nodes they are connected to, but may never rely upon their implementation. If all nodes are black-box by design, all participants of a transaction are either the initiator of the transaction or directly connected (adjacent) with the initiator. Nodes that are indirectly connected with the initiator can, by definition, not participate in a transaction driven by the initiator, since their existence is unknown to the initiator. Note that any participant of the transaction can in turn initiate new transactions in response to a message they process. These *sub-transactions*, however, are not known to the original initiator.

This black-box property is exploited by the concept of *tranquility*, which was introduced as an appropriate status for updatability:

A node is in a tranquil status if:

(i) it is not currently engaged in a transaction that it initiated.

(ii) it will not initiate new transactions.

(iii) it is not actively processing a request.

(iv) none of its adjacent nodes are engaged in a transaction in which this node has already participated and might still participate in the future.

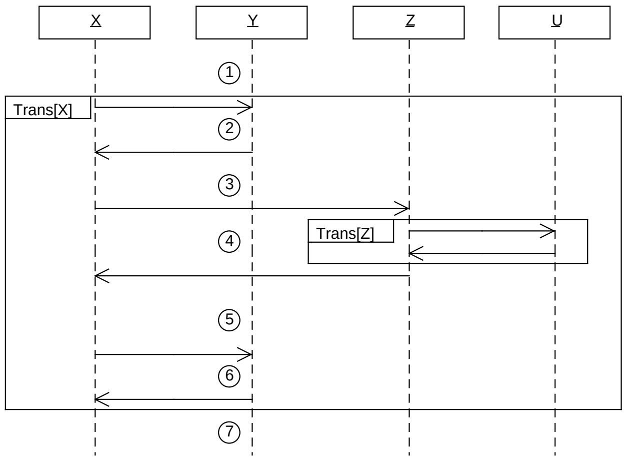

Figure 1: A Transaction being executed

Consider the transaction depicted in Figure 1. The transaction is initiated by component X, which directly communicates with Y and Z. As a consequence of this communication, the latter initiates a sub-transaction with component U. As component Y is directly involved in the transaction with X, it is only in tranquil state at points 1 and 7. U, on the other hand, is tranquil at all points except 4.

The notion of transactions can be used here, since it is an explicit concept in component-based architectures. Each service offered by a component is implemented as a transaction, which may involve an arbitrary number of other components in order to fulfill sub-transactions. The participants of a transaction are also made explicit in component-based architectures, as a component may only solicit another component via a connector between them.

In the next section, we will outline the implications of applying tranquility to plain objects, in the absence of a component-based architecture.

3. COMPONENTS VERSUS OBJECTS

Tranquility was originally established at the level of software components. However, most software systems are not designed on top of a component-based architecture, limiting the applicability of the approach. In order for tranquility to be applicable at the object level, two crucial issues need to be tackled.

The first issue follows from the fact that objects are no black box entities. Indeed, many programming languages allow for reflection, exposing an object's (or class) implementation details [12]. Things are even worse in the case of intercession, which is a special case of reflection allowing for the actual modification of an object's behavior. Therefore, the assumption that objects do not rely on the implementation details of other objects is not necessarily true.

The second issue has to do with transactions. In component-based systems, transactions are an explicit concept, known to the developer. When implementing a service in a certain component, the developer effectively describes a transaction. Moreover, since a component's external communication is conducted via so-called ports and connections, it is statically determined which components take part in such transaction. Hence it is possible at runtime to verify the requirements for tranquility, as listed in Section 2.
In typical object-oriented systems, there simply is no high-level notion of transaction being used.

Components can be considered singleton entities which continually process messages they receive in their message queue. Processing a message results in the invocation of a service, and hence implies the execution of a transaction.
Essentially, we propose to regard objects as mini-components with similar properties. Indeed, objects can also be viewed as entities which continually process messages, although depending on the degree of concurrency in the system, the length of the message queue may vary.

As for introducing transactions, there is a tradeoff between annotating the source code and simply automatically regarding every method implementation to be a transaction. While the performance cost of the former is clearly lower, it may not always be possible to modify the system source code. An alternative solution might be to mark specific methods as transactions, which could be achieved without the source. While introducing transactions in itself is not that hard, determining which objects participate in them is a more complex matter, as this can not be statically determined. Due to polymorphism, it is not even certain exactly which types are involved. Indeed, another difference between components and objects is that components are singletons, whereas objects typically belong to a certain type, which may be represented as a class, for example. Hence, a dynamic bookkeeping mechanism needs to be in place, which essentially keeps track of each object's acquaintances, i.e., those objects which can be accessed directly.

Given this mapping, sufficient information is available for the tranquility concept to be introduced: When an object is in tranquil state, it is no longer allowed to process any messages. If we assume that reflection (including intercession) occurs via message sending as well, this is also disallowed. Problems with reflection would occur, for example, if an object A obtains information on the internals of another object B and then processes this information after B has been replaced, rendering the obtained information obsolete. However, as long as retrieving and processing the information occurs within a transaction, it is guaranteed that the B object will not be replaced unless it is no longer used further on, in which case the problem does not occur. Indeed, if A does not address B anymore after having obtained the information it needs, it should not care whether or not B is replaced afterwards.

A final difference between components and objects is the differentiation between classes and instances. While in component-based systems there typically is only a single instance of a component, this usually is not the case for class-based systems in which many instances of the same class co-exist. In a class-based system, it may be desirable to update only one particular instance of a class, all instances or perhaps even a particular selection of instances. The most likely of these cases is to replace all instances of a class, but it may prove interesting to modify the behavior of only one instance, which would allow, for example, the optimization of the instance's behavior towards the client objects that are using the instance. For this reason, we aim to support both cases.

In the next section, we illustrate our proposal by means of a concrete scenario.

4. A SIMPLE EXAMPLE

In order to verify the practical applicability of tranquility on the level of objects, we hereby sketch the implementation of a framework that is capable of putting objects in a tranquil status upon demand. To do so, the framework simply observes messages between the objects until tranquility is observed, after which it blocks all incoming and outgoing messages by that object[1]. The framework is similar to the one presented in [21]. In comparison to that framework, however, this one requires additional mechanisms to (a) shield the behavior of objects and (b) keep track of the objects that are in a transaction and (c) to resolve some smaller issues.

[1]In some cases, tranquility is not reachable [21]. We do not elaborate on such situations in this paper

```
1  class Client {
2    PDFDownloader downloader = new PDFDownloader();
3    public PDF getRemotePDF(URL url) {
4        downloader.connect(url);
5        return downloader.getPDF();
6    }
7  }
8
9  class PDFDownloader {
10   TCPSocket myConnection = new TCPSocket();
11   public void connect(URL url) {
12       myConnection.connect(url);
13   }
14   public PDF getPDF() {
15       return new PDF(myConnection.getTCPData());
16   }
17 }
18
19 class TCPSocket {
20   public void connect(URL url) {...}
21   public Stream getTCPData(){...}
22 }
23
24 class UDPSocket {
25   public void connect(URL url) {...}
26   public Stream getUDPData(){...}
27 }
```

Figure 2: Java code of the PDF downloader

Before we demonstrate these extra difficulties, we first introduce an example application that will be adapted dynamically. Consider a simple system that is capable of downloading PDF documents over a network. The source code of this application is listed in Figure 2. The application consists of four classes: `Client`, `PDFDownloader`, `UTPSocket` and `UDPSocket`. A client instance can download a PDF document from a given URL. This behavior is implemented by the `getRemotePDF` method in the `Client` class. The method first creates a connection to a `PDFDownloader` instance and then uses this connection to download the PDF document (the `getPDF` method). The `PDFDownloader` instance uses a `TCPSocket` instance for transferring the PDF to the client in a way that is specific to the TCP data transfer protocol. Figure 3 presents the sequence diagram of the application.

Consider an evolution scenario that requires an adaptation of the `PDFDownloader` so that it uses a `UDPSocket` instead of a `TCPSocket` for transferring data to the clients. This update requires two modifications within the `PDFDownloader class`. Firstly, line 10 of Figure 2 needs to be replaced by
`UDPSocket myConnection = new UDPSocket();`
Secondly, line number 15 must be changed to
`return myConnection.getUDPData();`
In Figure 3, there are 11 different moments at which these changes can be applied to the running application. According to the definition of tranquility, the `PDFDownloader` instance is tranquil at moments 1, 2, 3, 4, 5 and 11 while the `PDFDownloader class` is tranquil at the moments 1, 5 and 11. Note that one can opt for updating only the behavior of a specific instance of the `PDFDownloader` or for updating the behavior of all `PDFDownloader` instances. The choice of the latter requires all of the instances to reside in a tranquil state, and consequently requires a mechanism to retrieve all the instances (like in [12]). We assume that such mechanism

can be implemented by means of introspection. The former choice requires a mechanism that allows one instance to have a proper behavior. There are already some approaches that allow this [15].

The application of this update on the `PDFDownloader class` at period 5, also requires an adaptation of all the instances of that class. Omitting this will result in a system that is inconsistent, as the new version of the `getPDF()` assumes that the connection is of the `UDPSocket` kind. Such consistency issues depend on domain knowledge and fall out of the scope of this paper, and we refer to [19] for more information. Likewise, the actual application of dynamic updates is out of scope, and we refer to approaches like [10, 16, 7]. For the remainder of this paper, we simply assume that the updates can be applied, and that the developer makes sure that the updates leave the software system in a consistent state. We get back to both assumptions in Section 6.

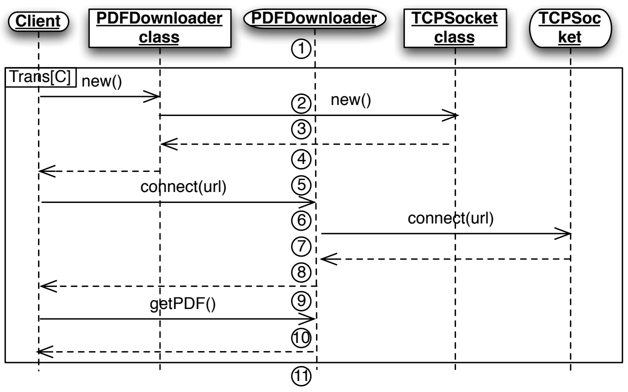

Figure 3: Sequence diagram of a PDF downloader application

5. IMPLEMENTATION

In Section 3, we already put forward two extra difficulties that had to be dealt with when shifting tranquility from the component to the object level. The first difficulty, is that in object-oriented systems, the software units are no longer considered to be black box. This brings along that the behavior of objects is not completely hidden away and that objects could still be solicited even when they are already put in a tranquil state. We plan to resolve this by (a) encapsulating the fields of every object [6] and (b) wrapping every object in such a way that "all" messages that are sent to an object (even the messages related to introspection and intercession) are first received by the wrapper, before they are propagated to the wrapped object [3]. Note that this technique does not bring along black box objects, but does allow one to keep an object tranquil, no matter the reflective capabilities.

The second difficulty mentioned above involves the support for transactions. In component-based systems, the communication between the components is made explicit within the code. In the component-based tranquility framework, this information is used to decide whether or not a certain component was involved in a transaction that was initialized by an adjacent component. This information is not available in object-based systems, as the communications between objects are achieved implicitly. In order to overcome this, we propose to apply a dynamic bookkeeping technique that is based on piggy-backing [22]. This basic idea is that the sender of a message always sends along a reference to itself. That way, the receiver object knows that it is currently in a transaction with the sender object. Upon termination of a transaction, the object sends an extra message to all the objects that participated in the transaction to tell them the transaction is over.

In addition to these two challenges, there are several other issues that we have to deal with, as we work with objects instead of components. For instance, every object will require a queue, in which it can store the messages that are sent to it while it is being updated. Moreover, in order to modify the interface of a method of one object (for instance the type signature or the amount of parameters), one does not only require that object to be tranquil, but also all the callers of that method. This is not trivial, as communications are not explicit in object-oriented systems. One way to resolve this issue in a safe way, is by applying a conservative approach in finding all the possible callers of a certain method (as it is implemented in the Eclipse IDE [5]). Other approaches – such as [23] – use approximation techniques in order to find less candidate callers. In our situation, however, a conservative approach might be more suitable.

We intend to implement tranquility at the object-level using the delMDSOC virtual machine model [8]. This machine model is primarily aimed at providing support for modularizing crosscutting concerns, which makes it, for example, suited for implementing aspect-oriented programming. Tranquility, often to be ensured for many objects and/or classes across the running system, is a crosscutting concern that needs to be deployed at runtime. As delMDSOC also supports the dynamic deployment of crosscutting concerns, it may serve as a suitable target platform. delMDSOC provides a prototype-based object-oriented environment and supports the actor-based model of concurrency [1, 9]. Advantages of that model include that it provides a message queue to every actor and that it supports concurrency. As the model supports message passing through the use of delegation, it is easily possible to modify an object's behavior by intercepting the messages sent to it. This mechanism can also be used to handle all bookkeeping in separate objects and to update either a particular instance or all instances of a particular class.

6. RELATED WORK

In previous work, we already stated that DSU incorporates several pitfalls [4]. This paper only focusses on one of them: the "program consistency and deactivation". Other pitfalls in DSU, such as the "introduction of new code" or the "State transfer and consistency" are not targeted by this paper. For related work that targets state consistency, we refer to [20] and [19]. For related work that targets the actual introduction of new code, we refer to approaches like [10], [16] or [7].

The idea of using a combination of wrapping and message queuing has been used before in the context of DSU. Examples of approaches that used those techniques include [17],

[14], [2] and [13]. A commonality between these approaches is that they all achieve *quiescence* and not *tranquility*. Although quiescence is proven to be sufficient for ensuring state consistency [11], achieving it causes serious disruption in the application which is updated due to the large number of nodes that need to be passivated [21]. By using tranquility instead of quiescence, one can overcome this drawback [21]. Note that we did not elaborate on the fact that tranquility is not always reachable. For that, a detection and fall-back mechanism – such as the one explained in [21] – needs to be included.

7. CONCLUSIONS AND FUTURE WORK

This paper has applied the notion of tranquility to objects, eliminating the requirement of a component-based architecture. In order to do this, objects are essentially considered to be mini-components, with their own message queue and transaction at method implementation granularity. We illustrated this approach by means of a concrete scenario and sketched a possible implementation. In comparison to the implementation of tranquility for component-based systems, this implementation requires additional mechanisms to shield the behavior of objects and keep track of the objects that participate in a transaction.

Future work includes the validation of our approach. We envision a of a proof-of-concept implementation accompanied by several benchmarks in order to measure possible performance overhead. An implication of using a wrapper-based approach is that the process of looking up messages becomes more expensive. This however can be countered using lookup caching strategies. The remaining overhead can be attributed to performing bookkeeping and the actual tranquility checks. In case benchmarks would show that our approach does have a large impact on performance, this impact can be reduced by manually specifying the transactions that should be considered instead of regarding every method call as a transaction, as indicated in Section 3. On a similar note, the notion of tranquility is only useful when an application update needs to be applied; this implies that it would be interesting, performance-wise, to allow for disabling tranquility checking when it is not needed.

8. REFERENCES

[1] G. Agha. *Actors: a model of concurrent computation in distributed systems.* MIT Press, 1986.

[2] A. Baumann, J. Appavoo, R. W. Wisniewski, D. D. Silva, O. Krieger, and G. Heiser. Reboots are for hardware: challenges and solutions to updating an operating system on the fly. In *Proceedings of the USENIX Annual Technical Conference*, pages 1–14, 2007.

[3] M. Büchi and W. Weck. Generic wrappers. In E. Bertino, editor, *Proc. ECOOP 2000*, pages 201–225. Springer Verlag, 2000.

[4] P. Ebraert, Y. Vandewoude, Y. Berbers, and T. D'Hondt. Pitfalls in unanticipated dynamic software evolution. In *Proceedings of the Workshop on Reflection, AOP and Meta-Data for Software Evolution*, pages 41–49, 2005.

[5] Eclipse Corporation. Eclipse, 2010.

[6] M. Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999.

[7] A. R. Gregersen and B. N. Jørgensen. Dynamic update of java applications—balancing change flexibility vs programming transparency. *J. Softw. Maint. Evol.*, 21(2):81–112, 2009.

[8] M. Haupt and H. Schippers. A machine model for aspect-oriented programming. In *ECOOP 2007 - Object-oriented Programming, 21st European Conference, Berlin, Germany, July 30 - August 3, 2007, Proceedings*, volume 4609 of *Lecture Notes in Computer Science*, pages 501—524. Springer, 2007.

[9] C. Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd international joint conference on Artificial intelligence*, pages 235–245, Stanford, USA, 1973. Morgan Kaufmann Publishers Inc.

[10] M. Hicks. *Dynamic Software Updating.* PhD thesis, University of Pennsylvania, 2001.

[11] J. Kramer and J. Magee. The evolving philosophers problem: Dynamic change management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, November 1990.

[12] P. Maes. *Computational Reflection.* Phd thesis, Artificial Intelligence Laboratory, Vrije Universiteit Brussel, 1987.

[13] I. Neamtiu, M. Hicks, J. S. Foster, and P. Pratikakis. Contextual effects for version-consistent dynamic software updating and safe concurrent programming. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 37–49, New York, NY, USA, 2008. ACM.

[14] I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol. Practical dynamic software updating for c. *SIGPLAN*, 41(6):72–83, 2006.

[15] J. Noble, A. Taivalsaari, and I. Moore, editors. *Prototype-Based Programming: Concepts, Languages and Applications.* Springer, 1999.

[16] M. Pukall, C. Kästner, and G. Saake. Towards unanticipated runtime adaptation of java applications. In *APSEC '08: Proceedings of the 2008 15th Asia-Pacific Software Engineering Conference*, pages 85–92, Washington, DC, USA, 2008. IEEE Computer Society.

[17] C. Soules, J. Appavoo, K. Hui, D. D. Silva, G. R. Ganger, O. Krieger, M. Stumm, R. W. Wisniewski, M. Auslander, M. Ostrowski, B. Rosenburg, and J. Xenidis. System support for online reconfiguration. In *Proceedings of the USENIX Annual Technical Conference*, 2003.

[18] C. Szyperski. *Component Software : Beyond Object-Oriented Programming.* Addison-Wesley, January 1998.

[19] Y. Vandewoude and Y. Berbers. Component state mapping for runtime evolution. In *In Proceedings of the 2005 International Conference on Programming Languages and Compilers*, pages 230–236, Las Vegas, Nevada, USA, June 2005.

[20] Y. Vandewoude and Y. Berbers. Deepcompare: Static analysis for runtime software evolution. Technical Report CW405, KULeuven, Belgium, Februari 2005.

[21] Y. Vandewoude, P. Ebraert, Y. Berbers, and

T. D'Hondt. Tranquillity: A low disruptive alternative to Quiescence for ensuring safe dynamic updating. *IEEE Transactions on Software Engineering*, 33(12), 2007.

[22] J. L. Weiner and S. Ramakrishman. A piggy-back compiler for prolog. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 288–296, New York, NY, USA, 1988. ACM.

[23] R. Wuyts. Roeltyper, a fast type reconstructor for smalltalk. Technical report, Université Libre de Bruxelles, 2005.