Invasive patterns: aspect-based adaptation of distributed applications

Luis Daniel Benavides Navarro, Mario Südholt,
Rémi Douence, Jean-Marc Menaud

OBASCO group, EMN-INRIA, LINA
École des Mines de Nantes, Nantes, France
{lbenavid,sudholt,douence,menaud}@emn.fr

Abstract. Software patterns are frequently used as a software development tool in sequential as well as (massively) parallel applications but have been less successful in the context of distributed applications over irregular communication topologies and heterogeneous synchronization requirements. In this paper, we argue that lack of flexibility of pattern definitions is a major impediment in distributed environments, especially legacy contexts. We propose *invasive patterns* that support the modular definition and adaptation of distributed applications in the presence of complex pattern-enabling conditions. Invasive patterns are concisely defined in terms of two abstractions: aspects (in the AOP sense) for the modularization of crosscutting enabling conditions, and groups of hosts for the definition of patterns over complex topologies. Concretely, we motivate the need for invasive patterns in the context of JBoss Cache, introduce the concept of invasive patterns and briefly discuss corresponding language support as well as an implementation.

1 Introduction

Software patterns have proven a versatile tool for program development, be it for the development of application designs [6], architecture descriptions or program implementations [4]. Design patterns have been very successful in the domain of sequential, in particular object-oriented applications and for the derivation and implementation of massively parallel algorithms [8, 3].

However, pattern-based approaches have been much less successful in the domain of distributed programming, in particular if they are not defined over regular communication topologies and subject to heterogeneous synchronization constraints. In this paper, we argue that lack of flexibility of pattern definitions is a major impediment in distributed environments, especially legacy contexts. Frequently, applications of patterns in distributed contexts depend on information on the execution state that is not directly available at the point where the pattern is to be applied but has to be (invasively) accessed elsewhere. The underlying pattern-based architecture can only be made explicit using new (grey-box) software composition techniques.

In this paper we propose a notion of *invasive patterns* for distributed programming. Such patterns extend well-known regular computation and communication mechanisms by a built-in abstraction for access to non-local state that is necessary to enable invasive pattern applications in distributed applications. We provide evidence that techniques from Aspect-Oriented Programming (AOP) [1] can be harnessed to provide structured access to such non-local state, thus complementing recent evidence that AOP is useful as a support technology for sequential pattern-based applications.

2 Motivation: patterns in JBoss Cache

As a motivating example for the problems in applying pattern in distributed (legacy) applications, we consider the implementation of two phase commit transactions in the current JBoss replicated cache framework [7].

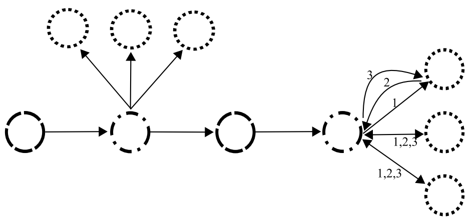

Fig. 1. Transaction handling under replication in JBoss Cache

Transaction handling under replication in the current production version (1.4) of JBoss Cache can be described *in abstract terms* as the architecture shown in Fig. 1a, *i.e.*, a pipeline pattern for transaction control (whose parts are represented by the dashed circles) and farm patterns (dotted circles) for replication actions.

More concretely, the transaction concern can be conceptually structured into two parts, locking of the tree structure that JBoss Cache uses as its main caching data structure and the two phase commit protocol between nodes. The transaction is triggered by a specific method call represented by the first node in the above figure. Then successive calls to the basic cache manipulation methods `get`, `remove` and `put` and the information is stored for further replication in other nodes. When a particular value is not found in the cache, the cache asks for the value from a group of selected neighboring caches, its so-called buddies. This interaction is shown by the three edges starting in the second node of the figure and ending in three other nodes. Once the end of a transaction is reached, the originating cache engages a two phase commit protocol. In such a protocol the originating cache sends a *prepare* message with the transaction control information (edges numbered 1 in the right part of the figure), followed by answers from all buddies stating agreement or non agreement (edges numbered 2). Finally, the originating cache sends a final commit or a rollback message depending on the answers it received (edges numbered 3).

However, while this distributed algorithm is nicely represented using patterns on an abstract level, these *patterns are not made explicit in the implementation*

```
 1 //----- Piece of code in the invoke method of class DataGravitation
 2 public Object invoke(MethodCall call) throws Throwable{
 3 ...
 4 if (!isTransactionLifecycleMethod(m)){
 5         if (isGravitationEnabled(getInvocationContext())){
 6 ...
 7 else{
 8 try{
 9  switch (m.getMethodId())
10   {
11     case MethodDeclarations.prepareMethod_id:
12     case MethodDeclarations.optimisticPrepareMethod_id:
13          Object o = super.invoke(m);
14         doPrepare(getInvocationContext().getGlobalTransaction());
15          return o;
16     case MethodDeclarations.rollbackMethod_id:
17          transactionMods.remove(
18                  getInvocationContext().getGlobalTransaction());
19          return super.invoke(m);
20     case MethodDeclarations.commitMethod_id:
21          doCommit(getInvocationContext().getGlobalTransaction());
22          transactionMods.remove(
23                    getInvocationContext().getGlobalTransaction());
24          return super.invoke(m);
25   }} catch (Throwable throwable){ ...
```

Fig. 2. Tangling of two phase commit (2PC) in the DataGravitation replication class

of JBoss Cache due to the scattering and tangling of code these functionalities are subject to (as has been analyzed for the sole replication functionality in [2]).

Figure 2 shows a piece of code of the *invoke* method in the `DataGravitation` class that is responsible for buddy replication and shows some of the crosscutting of transaction code JBoss Cache is subject to. This code presents a common idiom for transactions in the JBoss Cache implementation (see lines 9 to 24) that is often tangled with other functionalities. In this case the class `DataGravitation` was supposed to control the buddy replication concern and not the transactional behavior, the latter being handled by a specific transaction filter in JBoss Cache. This idiom is scattered over many places in the implementation. We have found 93 places where this switch statement is used and more than 28 places where it is used in the context of replication operations implying a farm-like communication between caches The class `DataGravitation`, *e.g.*, farms out data in its method `doCommit` that is called in the code excerpt. Hence, distributed patterns are triggered by complex enabling conditions in the code of JBoss Cache, which impedes a pattern-centric implementation of the code.

3 Invasive patterns

In order to be able to make explicit distributed patterns in crosscutting contexts as discussed above, two basic issues have to be addressed: (i) support for a set of basic distributed communication and computed patterns that may be composed with one another and (ii) support for structured invasive access to information that is not present where the communication itself occurs.

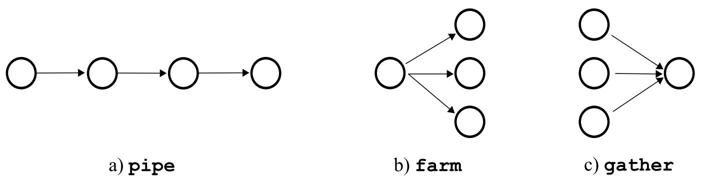

Fig. 3. Basic patterns

There is a very large choice of potential basic architectural patterns for distributed programming, *e.g.*, publish-subscribe relationships [5], skeleton-based approaches [3], or more recent work on patterns for grid-based systems [4]. Since one of the main goals of this paper is to investigate how patterns can be complemented by a notion of invasive access we consider here the three most basic patterns `pipe`, `farm` and `gather`, see Fig. 3 (where circles denote calculations that possibly take place on different hosts and edges denote communication).

In order to account for crosscutting enabling conditions for patterns, such as accesses to the transactional context in JBoss Cache as described before, that is not available at the point where the pattern itself is to be applied, we provide a new notion of *invasive patterns.* Aspect-Oriented Programming [1] seems a promising approach for the modularization of such patterns along with their corresponding data accesses.

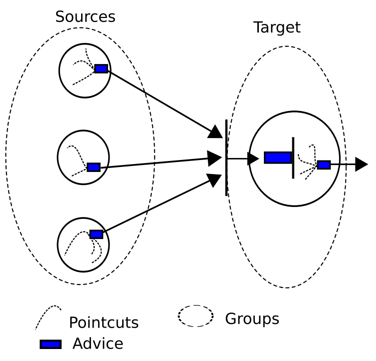

Fig. 4. Invasive gather pattern

We substantiate this idea in this paper by extending basic distributed patterns with a notion of aspects to modularize such crosscutting accesses. The resulting notion of patterns is illustrated in Fig. 4 for the case of the `gather` pattern. On the three nodes on the left hand side, different pointcuts (represented by dashed lines) are used to access information that is then prepared by "source" advice (represented by the filled rectangles) to be sent to the right hand side node. Once all relevant data has been passed to the right hand side node, a "target" advice is used to integrate a (or trigger a new) computation involving that data on that node. Note that "target" and "source" are realized in terms of groups of hosts: the application can refer directly to those groups to abstract from the underlying topology in pattern definitions.

3.1 Implementation using AWED

We have implemented our pattern language by means of a transformation into the AWED language for aspects with explicit distribution [2]. This transformation exploits AWED's concepts of distributed aspects, remote pointcuts, remote advice and dynamic groups of hosts to implement invasive patterns. The invasive pattern shown in Fig 4, including the pattern groups `sources` and `target` as well as source and target advice along with the necessary synchronization between source and target, is expressed using AWED's advanced language features, in particular sequence pointcuts and a/synchronously-executed remote advice.

3.2 Invasive patterns for JBoss Cache

Invasive patterns allow to concisely express the pattern-based architecture for transaction and replication handling in JBoss Cache as shown in Fig. 1. Concretely, we have implemented support for transactions under replication with pessimistic locking and the two phase commit protocol.

```
 cacheGroup = {H1, H2, H3}
pipe([h], Atransac,
  farm(
    gather(
      farm([h], Aprepare, sync cacheGroup-[h]),Apresp, [h]), Acommit, cacheGroup-[h]));
```

Fig. 5. Pattern-based definition of JBoss Cache two phase commit

The corresponding solution is formulated in terms of a nested composition involving four patterns, see Fig. 5. First, we apply a pipeline pattern to encapsulate the transaction start. Once a commit is encountered, a *farm* pattern is used to farm-out the prepare phase of the two phase commit protocol. Then, a *gather* pattern is used to collect the answers from the involved buddy caches. Finally, after all answers have been received we use again a *farm* pattern to distribute the final decision of commit or rollback. The code in the figure defines this algorithm for three replicated caches. Note that the implementation is parametrized over a cache group of three hosts and the protocol can be triggered from any of the three caches. Once the triggering host is fixed, the expression *cacheGroup-[h]* represents the group of caches without the triggering one.

Invasive accesses required to make this solution work are provided by the involved aspects `Atransac`, `Aprepare`, `Apresp` and `Acommit`. Figure 6 presents the pattern-defining aspect `Aprepare` that realizes the prepare phase of the two phase commit protocol. Occurrences of calls to the `prepare` method are matched (see the pointcut definition on lines 3 to 5) and the target advice (see target advice definition, lines 9 to 14) executes a prepare phase followed by the invocation of an agreement or disagreement method, depending of the answer of the buddy caches. Furthermore, this aspect does not replicate if the transaction occurs in the control flow of a prepare phase, *i.e.*, replication occurs only in the top-level call to the prepare phase not nested ones.

```
aspect Aprepare{
  org.jboss.cache.TreeCache tc = CacheRegistry.getInstance().getCache();
  around(DataStorage d, String txId):
       call(* PrepareHelper.send(..)) && args(d,s) &&
     !cflow(call(TransactionManager.prepare(..)));
  // source advice
  {proceed();}
  // target advice
  { TransactionManager tm = TransactionManager.getInstance();
    PrepareHelper ph = new PrepareHelper();
    try {
      tm.prepare(d, txId, tc);
      ph.respAgree(txId);
    } catch(Exception e){ph.respNotAgree(txId);}}}
```

Fig. 6. 2PC invasive aspect triggers the two phase commit protocol

4 Conclusion

In this paper we have introduced the notion of *invasive patterns* that allow better modularization of crosscutting enabling conditions of traditional distributed communication and computation patterns. In the context of JBoss Cache, we have motivated that such crosscutting is a major impediment for the use of patterns in real-world distributed applications and have given evidence how invasive patterns help bridge the gap between pattern-based architectures and implementations.

We have sketched language support for invasive patterns and briefly discussed an implementation of this language based on AWED, a system for explicitly distributed AOP. We are currently working on augmenting the expressive power of invasive patterns, their optimized implementation and their formal properties.

References

1. M. Akşit, S. Clarke, T. Elrad, and R. E. Filman, editors. *Aspect-Oriented Software Development.* Addison-Wesley Professional, September 2004.
2. L. D. Benavides Navarro, M. Südholt, W. Vanderperren, B. De Fraine, and D. Suvée. Explicitly distributed AOP using AWED. In *Proceedings of the 5th ACM Int. Conf. on Aspect-Oriented Software Development (AOSD'06).* ACM Press, March 2006.
3. M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation.* MIT Press, 1989.
4. J. Easton et al. *Patterns: Emerging Patterns for Enterprise Grids.* IBM Redbooks. IBM, June 2006. `http://publib-b.boulder.ibm.com/abstracts/sg246 682.html`.
5. P. Th. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131, June 2003.
6. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley, 1994.
7. JBoss Cache home page. `http://labs.jboss.com/jbosscache`.
8. S. Siu, M. De Simone, D. Goswami, and A. Singh. Design patterns for parallel programming. In *Proc. of Parallel and Distributed Processing Techniques and Applications (PDPTA)*, pages I, 230–240. C.S.R.E.A. Press, August 1996.