The Propagate Cache Updates Pattern

Kyle Brown
Senior Technical Staff Member
IBM Software Services for WebSphere

Introduction

This pattern belongs to an evolving pattern language called "Patterns of System Integration with Enterprise Messaging". The pattern language describes how systems are developed using the capabilities of Enterprise Messaging software like IBM MQ Series, Sonic Software's SonicMQ, or Microsoft Message Queue.

The place of this pattern in the language

This pattern is part of a largely undeveloped section of the pattern language on application architectures that use messaging. These patterns are more finely-grained, and less generally applicable than the other patterns in this language, documented in [Woolf] and [Hohpe]. Other patterns in this section that have not yet been published include *Logging to a Queue*, where a logging infrastructure is built using Enterprise Messaging and *Local and Remote Update*, where we discuss how to keep information held both in a local (distributed) database and a remote (mainframe) database in synchronization using Enterprise Messaging.

This pattern refers to many of the patterns described in [Woolf]. A diagram of some of the relationships that this pattern has with other patterns in the language is shown below (Figure 1: Pattern Relationships):

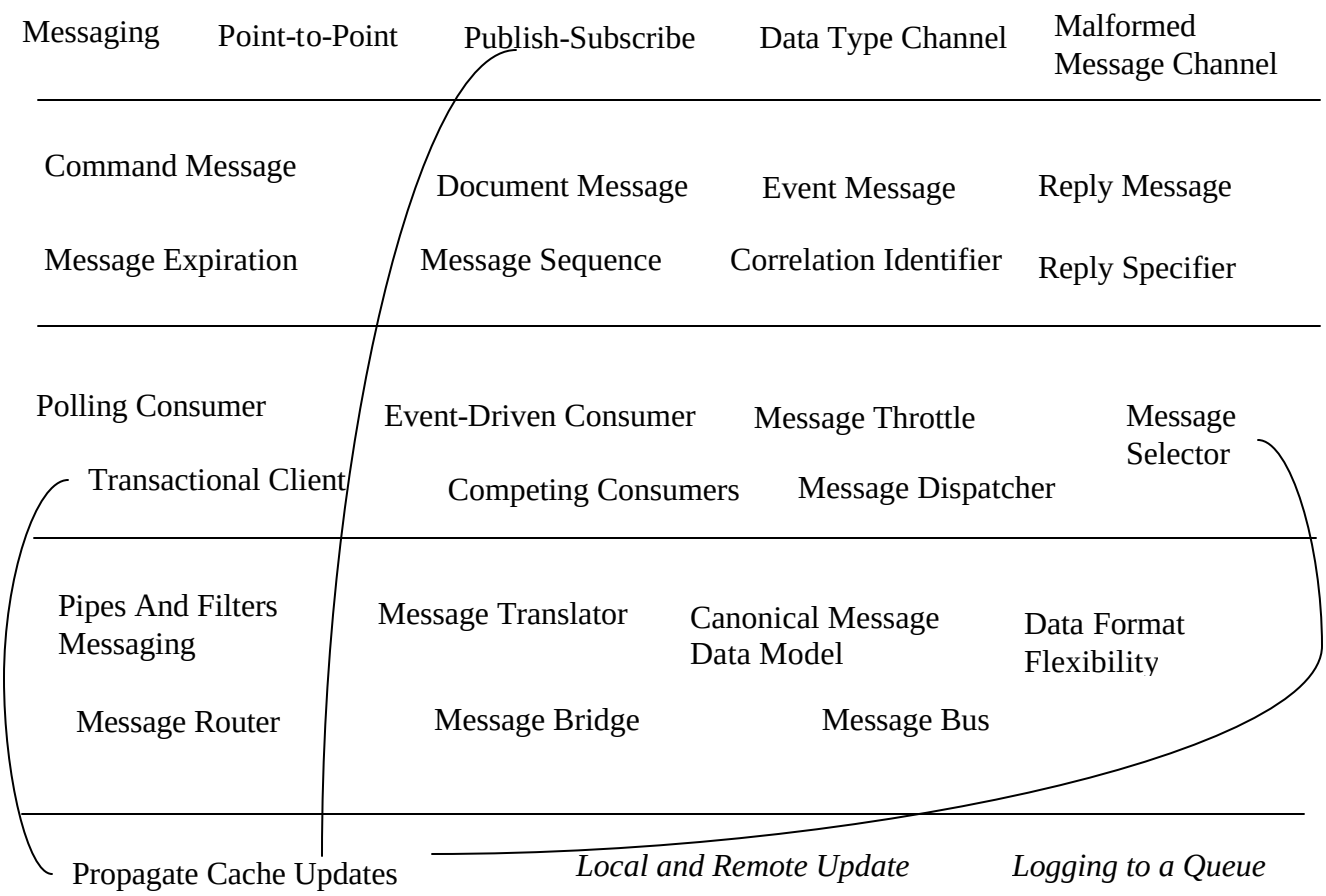

Figure 1: Pattern Relationships

Propagate Cache Updates

My application is distributed over several physical machines for scalability. It uses a database for object persistence, but many of the queries to the database take a long time to execute due to the complexity of the queries. My database queries cannot be further optimized, so it is impossible to gain more speed through database tuning approaches. I would like to cache my data on each machine; however, I cannot cache all of my data locally since the data does change, and the values in each cache will begin to differ from the database and each other over time.

How can I connect a set of distributed caches such that updates are propagated across the caches and the same values can be returned from queries on any of the caches?

Many systems are designed with a set of data caches to improve performance. For instance, in a system built using Enterprise Java Beans (EJBs) you may use Entity Bean Option A caching [EJB], or we keep value objects in memory in a singleton [Brown]. However, each of these options have the same drawback; for instance, in Option A Caching, once a CMP EJB is read from the database, the values held in memory do not change, even though the corresponding values in the underlying database may change.

Most distributed HttpSession schemes also are a type of distributed data cache. The similarity of each of these approaches has led to the recommendation of a specific API for caching, the JCache API, as JSR-107 [JCache]. Unfortunately, the cache is not the "system of record' for most of this information. In almost all cases, the ultimate place where data is stored is in a database, thus creating a situation where the information in the database and the information in an in-memory cache can drift out of synchronization. When the database is updated by one machine, if a query is run against the (older) data in the cache on another machine it will return the wrong value.

Some systems have been built such that the database itself is responsible for updating the set of distributed caches. These systems use database triggers to force the update of each cache. The problem with this approach is that it is not standards-based and thus is not portable. Thus a system built using Oracle database triggers will not work if the database is changed to DB2 or SQL Server. Also, not every database (for instance some of the open-source databases like MySQL) supports advanced database features like triggers.

Thus, we need a way to force an update of each cache whenever an object is changed in any cache. Therefore:

Propagate cache updates using *Publish-Subscribe* messaging such that whenever a change is made to an object in one cache, that server will to notify all other caches that the object has been changed.

If a cache receives a notification it can choose to refresh its values from the database (shared) version at notification time, or it may simply mark it as "dirty" and read it back from the database whenever anyone asks for it. The structure of these solutions is shown below (Figure 2: Distributed Cache Update). Another option would be to update the cache from the message. However, this is not as desirable as reading from the DB since it would require object hydration from a message instead of from the database; this both complicates the messaging code, and also increases the amount of message traffic in the system as a whole since the entire object, rather than a notification, must be sent in the queue.

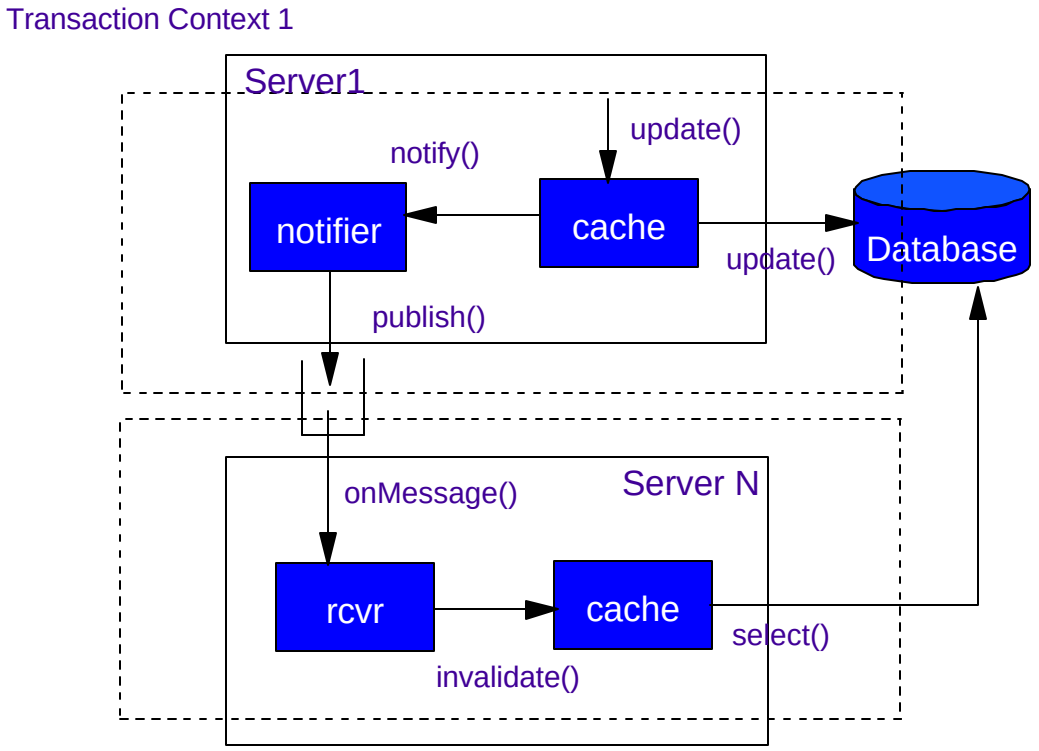

Figure 2: Distributed Cache Update

It is important to keep the granularity of the cache high such that the total number of messages flying across the messaging system is kept to a minimum. In many cases, this can be achieved by sending out only notifications about the "root" object of an object graph whenever any part of the graph changes. Within a transaction you hold off notification until all updates complete so that we can remove redundant messages. Likewise, it is desirable to have the “put” onto the queue be part of the same transaction as the update to the database (e.g., make the cache a *Transactional Client*) so that the state of the database does not diverge from the known state of the caches (Figure 2 shows the database update and the message being part of the same transactional context, labeled “Transaction Context 1”).

You can reduce the amount of unnecessary processing that each cache must perform in handling update messages for objects it does not contain by introducing multiple topics (one for each "root" type). The cache could use *Message Selectors* to filter out notifications about objects they are not interested in, but that does not reduce the total number of messages that are placed on the topic – it only reduces the amount of processing each client will perform.

It is also crucial that this solution only be used in cases where it is not crucial that all caches remain perfectly synchronized at all times. This is because the solution necessitates the use of at least two transactions; one on the "notifying" cache side, and another for each of the "notified" caches. This is shown in Figure 2 where the receiving end (Server N) is shown to be executing in a separate transactional context from the original transactional context. Thus, there can be a period of time while updating is occurring in which queries to one of the outlying caches can return stale data. There is also the possibility of undeliverable messages, incorrect update processing, and other situations that can render this solution less than 100% reliable. However, in most applications, so long as all final decisions depend solely upon the state of the database of record the unreliability of this solution can be tolerated.

This approach has been used successfully in commercial Java application server implementations. For instance, IBM WebSphere Application Server 5.0 uses this approach in synchronizing its HttpSession caches. Likewise, this is a feature of WebLogic Application Server 6.1. SpiritSoft sells a product called SpiritCache [SpiritSoft] that implements a JSR 107-compatible cache using this pattern that will work with many application servers.

Finally, this approach has been implemented in end-user applications for several large financial web sites by IBM Software Services. A specific implementation of this pattern restricted to EJB Entity Bean caching has been previously documented in [Rakatine] as the *Seppuku pattern*.

Bibliography

[Brown] Kyle Brown, Choosing the Right EJB Type, IBM WebSphere Developer's Domain, http://www7.software.ibm.com/vad.nsf/Data/Document2361?OpenDocument&p=1&BCT=66

[EJB], EJB 1.1 Specification, Sun Microsystems, http://java.sun.com/products/ejb/docs.html#specs

[Hohpe] Gregor Hohpe, "Enterprise Integration Patterns", http://www.enterpriseintegrationpatterns.com/

[JCache], JSR 107; JCache – Java Temporary Caching API, http://www.jcp.org/jsr/detail/107.prt

[Rakatine] Dimitri Rakatine, "The Seppuku Pattern", The ServerSide.com Newsletter #26, http://www.theserverside.com/patterns/thread.jsp?thread_id=11280

[SpiritSoft] SpiritCache overview, http://www.spiritsoft.com/products/jms_jcache/overview.html

[Woolf] Bobby Woolf and Kyle Brown, "Patterns of System Integration with Enterprise Messaging", submitted to the PLoP 2002 conference, http://www.messagingpatterns.com