Prevalent Systems: A Pattern Language

By Ralph E. Johnson and Klaus Wuestefeld
rjohnson.uiuc@gmail.com, mail@klaus.pro

Data often needs to be persistent to survive loss of power and to be accessible in the future. There are many ways to make data persistent, such as storing it in a local file system or storing it in a relational database management system. Each way has advantages and disadvantages. Prevalence is a way of achieving persistence that is fast and simple, easy to reason about and easy to implement. It can be extremely reliable and can work in all kinds of environments. Its main limitation is that data must all fit in memory, but when even a cell phone will have gigabytes of memory, many applications that once required a standard DBMS would be better if they used prevalence. Prevalence is well suited for object-oriented programming, since it lets the programmer think of the entire system from an object-oriented point of view. In fact, the prevalent system can be built using any paradigm, so it allows the programmer to chose the point of view that best suites the project. But its main advantage is speed. A prevalent system can provide the features of a database with the speed of main memory.

This paper describes the patterns *Prevalent System, Snapshot, Transactions and Queries, Transactions for User Interface, Transactions Based on OIDs, Transactions Based on Domain Names, I/O Outside, Replicated Server, and Short Transactions.*

Prevalent System: A system using prevalence consists of the *prevalent system* and *clients* that use it. The prevalent system is the data that needs to be persistent; it is a set of classes that are sometimes completely independent of the clients, but often at least partly shared. Clients communicate with the prevalent system by executing *transactions*, which are implemented by a set of transaction classes. These are examples of the Transaction design pattern[Gamma 1995]. Transactions are written to a *journal* when they are executed. If the prevalent system crashes, its state can be recovered by reading the journal and executing the transactions again.

The complete pattern consists of these four parts (prevalent system, clients, transactions, journal) and a prevalence manager.

Replaying the journal must be deterministic, so transactions must be deterministic. Although clients can have a very high degree of concurrency, the prevalent system is single-threaded, and transactions execute to completion. To ensure good response time, transactions should be short. The prevalent system has few constraints on its design, which means that it can be designed for performance and expressibility.

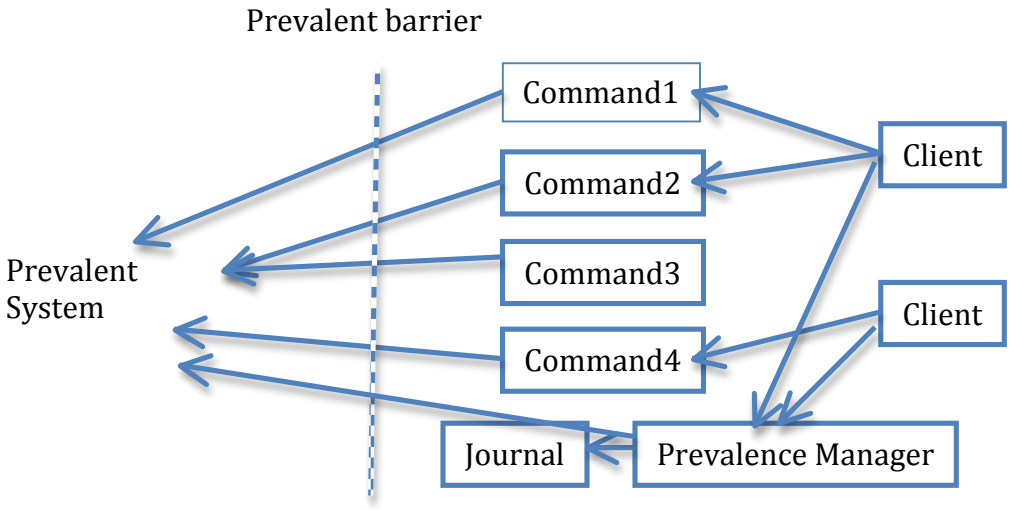

Fig. 1. Overview of prevalent system pattern

The transactions form a barrier between the prevalent system and the clients. This "transaction barrier" divides the part that is nondeterministic, concurrent, and has I/O (the clients) from the part that is deterministic, single-threaded and has no I/O (the prevalent system). Transactions must ensure that object identity inside the prevalent system is never used outside it. In other words, they must never allow pointers to objects to pass the transaction barrier. They can pass copies of objects, strings that give names to objects, or numbers that are IDs of objects, but they cannot pass direct references to objects.

The prevalent system must be deterministic, since this allows it to rerun the journal and produce the same results. This means that it should not include I/O (see *I/O Outside*), and it probably should not contain parallelism. Parallelism is a common source of non-determinism. It is possible to make parallelism deterministic, but it is easiest to just make the prevalent system single threaded. Another source of non-determinism is an overly aggressive optimizing compiler and floating point numbers, since floating point addition and multiplication are not associative. Finally, unordered data-structures like sets and maps can result in non-deterministic iteration.

One of the questions about the design of a prevalent system is what the transactions are like. How can transactions be designed to be short? There are several different ways to design transactions, such as "Transactions from human interface" and "Transactions based on OIDs"

Examples: The name "prevalent system" did not become popular until the release of the Java framework Prevayler[Wuestefeld 2001].

However, the pattern is much older than that. Smalltalk-80 (from 1980) uses a prevalent system for storing code [Goldberg 1984]. A Smalltalk "image" is a snapshot that contains both code (in the form of classes) and data. The "changes file" is a journal that contains each code change. A programmer who crashes the image can restart the most recent image and use a recovery tool to select which of the "recent changes" should be re-executed.

Other old examples?

Snapshot: To make recovering state faster, the prevalent system can be periodically saved to disk as a *snapshot*. Then, the state can be recovered by reading the snapshot and only re-executing transactions that happened after the latest snapshot.

Most prevalent systems use snapshots, but they are not always necessary. They are not necessary if it is not important to recover state quickly, and if the journal is never long. Also, if there are *replicated servers* then a new server can be created by copying the state of an existing server, and so the state of a server never needs to be saved to disk.

Transactions and Queries: Writing transactions into the journal has an overhead. Transactions that do not change the state of the prevalent system do not need to be replayed. So, an important optimization is to distinguish between transactions, which change the state of the system, and queries, which do not. Both are implemented by classes at the transaction barrier, i.e. queries are also examples of the Transaction design pattern. Transactions require exclusive access to the prevalent system, though queries can execute in parallel. Only transactions need to be saved in the journal.

Not writing queries in the journal saves a little time when the query is executed, and saves a lot of time when the journal is replayed. Queries can be more efficient if you have *replicated servers*.

*Transaction Barrier***:**

Transactions form a barrier between the prevalent system and the clients. They must ensure that object identity inside the prevalent system is never used outside it. Transaction arguments and transaction results cannot contain objects from inside the prevalent system. Suppose that a transaction were to create an object and return its identity to a client and then the client were to use that identity as the argument to a second transaction. These transactions could not be replayed, because when the first transaction was replayed, it would create a new object but the second transaction would still have the identity of the original object. How can we design the transaction barrier so that the prevalent system hides object identity?

Transactions for User Interface: Sometimes applications have a well-defined user interface that can be used to define transactions. Since the user interface communicates in terms of text and numbers, the transaction can, too. For example, consider a web application. Suppose that the transactions for this application are the set of HTTP get and post commands that the web server accepts. The fields of these commands are all text strings. The results of the commands are also text strings. So, the get and post commands can be easily stored in the transaction journal, since they do not refer to any objects, only strings. This is an example of *Transactions from User Interface*. It doesn't take much design work to decide on transactions, other than deciding which are read-only queries and which are real transactions.

Transactions based on OIDs. Suppose clients use many of the same classes as the prevalent system. This is often what happens with web applications built from standard web frameworks like Tomcat or Ruby on Rails. If the get and post commands of a web server were the transactions of a prevalent system then the web server itself would be inside the prevalent system. It would not be possible to use a standard web framework because the web server includes the I/O. Instead, these frameworks expect that they will call the prevalent system. In this case, transactions have to be designed to fit the application. If you aren't careful, transactions arguments and results might be objects from the prevalent system. The prevalent system would not be hiding object identity.

One solution is for the prevalent system to give every object a unique object ID (OID). Transactions (and queries) only return copies of objects, but they keep the same OID. When a transaction is executed, it can convert each argument with an OID to the original object, and if an argument is a new object (if it has no OID) then it can create a copy inside the prevalent system.

Transactions with domain names: Often a class requires that each instance have a unique name. For example, often a Customer object has a unique Customer ID, and an invoice has a unique Invoice Number. These names usually come from the problem domain. Then transactions can use these unique names to refer to objects in the prevalent system. Some of the objects in the prevalent system have unique names and some do not. For example, an Invoice usually has a set of detail lines, and they probably don't have unique names. If all transactions use domain names then transactions can only refer directly to objects that have unique names.

I/O Outside. The prevalent system should not have any I/O in it. I/O should be performed by clients, who then make calls on the prevalent system to record what they have done. Otherwise, the prevalent system is not deterministic, since it cannot ensure that the execution of a transaction always produces the same result. So, any transaction that needs to perform some I/O must be redesigned in such a way that the client can perform the I/O. Often this is done by splitting the transaction. For example, suppose the prevalent system kept the name of a file and a client needed to read this file and store its contents. Instead of having a single

transaction to read the file and store it, the client would have to first run a query to get the file name, then read the file itself, and then perform a transaction to store its contents. The transaction would contain the contents of the file, so replaying the transaction would allow the same value to be stored even if the file had been changed.

Replicated server. When a prevalent system crashes, it can be restored by loading the snapshot and then by replaying the transactions in the journal. However, if the snapshot is large or the journal is large then this can take a long time. One way to make recovery fast is to replicate the prevalent system on another computer. Clients should be running on different computers than the replicas of the prevalent system. When the main replica crashes, the clients can switch to a different replica.

It is easy to replicate a prevalent system. Assume one of the replicas is the "primary" and the others are "backup". Clients communicate with the primary, but it sends all transactions to the backups, which execute them immediately. If the primary fails, one of the backups can immediately take over.

When a snapshot becomes large, taking a snapshot can take a long time. Without replication, the prevalent system will probably have to pause during the snapshot. Ensuring a consistent snapshot without pausing is difficult, and it is more common to just use a backup replica to make the snapshot. The backup can accumulate transactions while it is performing the snapshot and then execute them once the snapshot is finished. So, the backup can make a snapshot without slowing down the primary.

Age of Empires and Starcraft both use prevalence for persistence and use replication to support multiple players. They support replaying a game (they keep a journal) and saving a game (taking a snapshot). When several players are simultaneously playing the same game, the state of the game is replicated and each player has a complete copy of the entire state. An action by one player must be broadcast to the other players, and each player sees the same sequence of actions as every other player. This works for internet games because the player actions are very small compared with the work done by the game.

Short transactions. A prevalent system executes one transaction (or query) at a time. To give good response time, all transactions should be short. If there is only one user then long transactions are not a problem, because the user will expect them to slow the system. But if there are many users and one executes a long transaction then the others will see the system pause until the transaction has finished.

What does "short" mean? It depends on your requirements. The LMAX system runs 6 million transactions per second on a commodity PC. This means that a LMAX transaction takes less than 166 nanoseconds on average. But this is extreme, and it

takes very careful engineering to keep transactions this short. Most system do not have such extreme performance requirements, and so

In practice, it is important to keep a record of transaction times so that developers can find long transactions and fix them. The prevalent system can easily time transactions and store their times in the journal.

If the prevalent system is replicated then long queries can be run on a replica. Some systems will detect long queries and run them on a special "long query" replica because users who run long queries don't expect a short response time. Queries only have to be run on a single replica but a transaction has to be run on all of them, so replication can increase throughput if most of the load is queries, but it will not help much if most of the load is transactions.

In general, if a transaction is too long, it has to be broken into a number of shorter transactions. Consider the process in a payroll system that sends paychecks to all employees. This might be done as one transaction, or one that first computes the paychecks and one that actually sends them. Neither approach is scalable; the more employees, the longer the transaction. The transactions would be shorter if the client would handle one employee at a time. So, the client process that sends paychecks to employees should generate them one at a time and each transaction would handle only a single employee.

Sometimes transactions can be made faster by caching and lazy evaluation. For example, suppose a transaction updates a value, and there are a hundred other values that depend on it. Recalculating each of those other values might take too long. However, the transaction could simply indicate that they need to be recalculated. Those values would then be recalculated the next time they were read. Recalculating those values would then count against the time budget of the transactions that read them, not the transaction that changed them. This would work if the transactions that read them were short and only used a few of the values. If one of the transactions read all hundred values, the "optimization" probably would not help.

References

[Fowler 2011] *The LMAX Architecture* by Martin Fowler, http://martinfowler.com/articles/lmax.html, 2011.

[Gamma 1995] *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Addison-Wesley, 1995.

[Goldberg 1984*] Smalltalk-80: The Interactive Programming Environment* by Adele Goldberg, Addison-Wesley, 1984. Chapter 23, System Backup, Crash Recovery, and Cleanup.

[Wuestefeld 2001] *Object Prevalence* by Klaus Wuestefeld, http://www.advogato.org/article/398.html, 2001