Prevalent Systems: A Pattern Language for Persistence

V October 16, 2014

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

Abstract:

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. It requires that data must all fit in memory and operations that change state must be deterministic. 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. This paper will describe a prevalent system as if it were object-oriented, but a prevalent system can be built using any paradigm. Its main advantage is speed. A prevalent system can provide the features of a database with the speed of main memory.

Introduction:

A prevalent system is an in-memory database that is simple to implement and can be very fast. Like other databases, it can provide atomic transactions to multiple users, and provide reliability in the face of system crashes. Unlike a RDBMS, a prevalent system does not usually come with a query language that can be used by non-programmers.

A prevalent system achieves speed by keeping all data in memory and by eliminating waiting. Once a transaction starts, it runs to completion. Only one transaction can run at a time. Some database systems achieve speed by parallelism, but a prevalent system usually minimizes parallelism. The clients can be parallel, but the prevalent system is not, though a *Replicated Server* has a form of parallelism.

This paper includes a number of patterns that go with *Prevalent System.* Recovery is often faster after a crash if a system uses *Snapshot.* By separating *Transactions and Queries,* queries can become faster. There are several alternatives for defining the interface for transactions, namely *Transactions for User Interface, Transactions Based on OIDs, and Transactions Based on Domain Names.* To ensure determinism and speed, keep *I/O Outside.* To make recovery extremely fast, use a *Replicated Server.* To improve response time, use *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 defined by a set of classes that are sometimes completely independent of the clients,

but often at least partly shared. However, the instances in the prevalent system are not shared with the clients. Clients must refer to proxies, or copies, but never to the instances stored in the prevalent system. The prevalent system is a database, and there should be a barrier between the prevalent system and its clients.

Clients communicate with the prevalent system by executing *transactions*, which are implemented by a set of transaction classes. These are examples of the Command 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. The prevalence manager is in charge of executing transactions and writing them to the journal. Typically, it writes them to the journal before executing them. If the prevalent system is replicated then the manager will be responsible for communicating with other replicas.

Replaying the journal must always give the same result, so transactions must be deterministic. Although clients can have a 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.

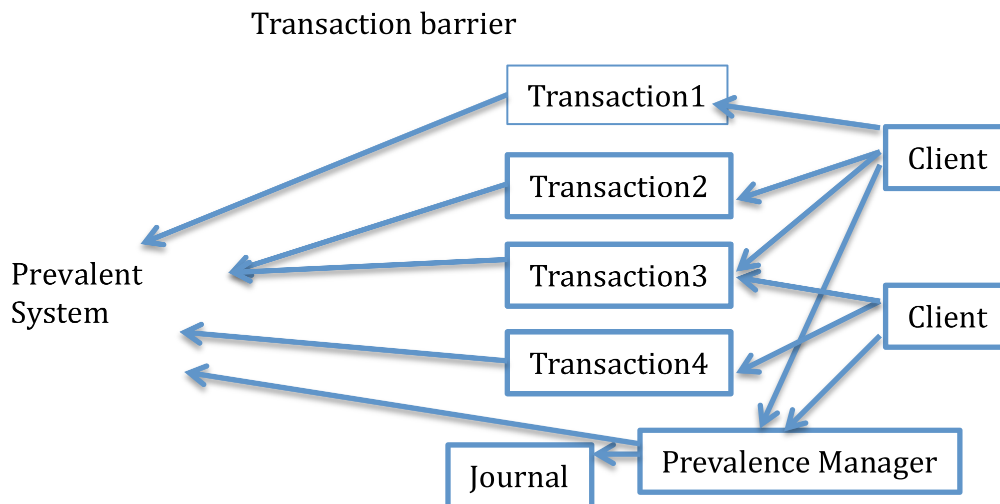

Fig. 1. Overview of Prevalent System pattern

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.

An early paper on using large memory showed the tremendous advantages of implementing a relational database as an in-memory database with a transaction journal and a single lock[Garcia-Molina 1984]. This design is the same as a prevalent system, except that it was not object-oriented.

Akka (since version 2.3.0) has support for "persistent actors. The state of an actor changes only when it receives a message. An Akka actor is made persistent by storing the messages in a journal. The system can also save the state of an actor with a *snapshot*. Akka implements actor migration by using the journal and snapshot to restart an actor on another server. [Akka 2014]

OrigoDB is a prevalence system for .NET. It is advertised as a in-memory database with blazing speed, high productivity, and the ability to define your data model in C#.

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.

One way to take a snapshot is to use a standard serialization mechanism, such as Java serialization. However, this ties the format of the snapshots to details of the implementation of the domain model, and refactoring the domain model can easily break the snapshots. If this approach is taken, snapshots need to be labeled with the version of the domain model and custom readers written when the version of the format changes. It might be easier to define a standard format for snapshots that does not depend on the details of the domain model and to modify the importer if the domain model changes enough.

When there is a lot of data, it can take a long time to take a snapshot. If the system does not run 24/7 then a snapshot can be saved when the system is not active. Another alternative is to have *replicated servers* and to use a replica to take the snapshot.

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 Command design pattern. Since queries do not change the state, they can execute in parallel. However, transactions require exclusive access to the prevalent system. It is theoretically possible to allow queries to execute in parallel until a transaction needs to execute. But a more common way to allow queries to execute in parallel is to provide *replicated servers* where each server executes all the transactions but a query only has to run on a single server.

Only transactions need to be saved in the journal. If there are many more queries than there are transactions then distinguishing between transactions and queries can reduce the number of transactions written to the journal and so reduce the cost of the journal.

The cost of writing a transaction in the journal varies. It can be more expensive to write a transaction to the journal than to execute it. The cost of writing a transaction to the journal depends partly on its arguments, because writing a transaction to the journal requires writing all of its arguments. Sometimes a transaction with a lot of arguments can be split into a query with a lot of arguments

and a transaction with a small number of arguments, and this will make the system faster because the query doesn't have be written to the journal.

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.

The main disadvantage of this approach is that the user interface is likely to change. Reading old journals is difficult if the format of the transactions changes. One way to solve this is to never change existing transactions. Instead, make a new transaction and deprecate the old ones.

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.

The main disadvantage of this approach is that refactoring the domain model can cause a different number of objects to be created, which can cause a transaction to produce an object with a different OID. This can make it hard to replay a journal after changing the domain model (the classes 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.

The main disadvantage of this approach is that transactions can only refer to objects with domain names. If it is necessary for transactions to refer to objects that don't have unique names, perhaps it is possible to give them names relative to the ones with unique names. For example, a detail line of an invoice might refer to the Invoice Number and then the line number.

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. One reason to remove I/O from the prevalent system is because input is not deterministic. Reexecuting a transaction that reads from a keyboard or a socket will probably give a different answer. A second reason is that I/O is slow, and prevalent transactions should be fast. By moving all I/O out of the prevalent system, we can ensure that transactions are fast. 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.

Moving I/O out of the prevalent system often requires 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.

An easily overlooked kind of I/O is communicating with the clock. For transactions to be deterministic, they must not read the clock. Instead, each transaction is labeled with the time it was originally executed (probably by the prevalence manager) and code inside the prevalent system must use the time of the current transaction instead of reading the clock. That way, re-executing the transaction will

give the same results. Libraries that access the clock directly have to be changed to use the transaction before they can be used in a prevalent system.

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. 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.

All replicas must execute all transactions in the same order. This means that a client can't just find the nearest replica and have it execute a transaction. Typically, transactions are sent to the primary and it gives sequence numbers to the transactions. The backup replicas will execute them in the order determined by the primary. This replication does not speed up transactions. However, replication can speed up queries. Since queries don't change the state of prevalent system, they can be run on any replica.

When a system becomes distributed, fault-tolerance becomes important. Ensuring that a set of replicas maintain consistent copies of the journal is an example of the "replicated state machine" problem. A popular solution to this problem is the Raft concensus algorithm[Ongaro and Ousterhout 2014], which has open-source implementations in a variety of languages.

When a snapshot becomes large, taking a snapshot can be slow. Without replication, the prevalent system will 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.

Croquet is an architecture for creating collaborative, shared 3D worlds over the internet[Smith et al 2003]. Each person in a Croquet world has an avatar in it, and

runs a replica of the world. Actions that they perform, such as moving their avatar or moving an object in the world, are transactions, and so must be communicated to every replica. Each Croquet world has its own serializer. All transactions go to the serializer for that world, which then broadcasts them to other replicas. However, rendering the 3D world on the screen is a query, and can be done independently by each replica. Rendering a 3D scene is computationally intensive, so it is important that it be done locally. Each action only affects a few objects, so it takes little data to communicate them. This is important because they have to travel to all the replicas. This provides good performance even over internet distances. For Croquet, replication provides performance by allowing the computationally intensive rendering to be done locally.

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[Fowler 2011]. 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 systems do not have such extreme performance requirements. Suppose you wanted a 10 ms response time when you had 100 active clients. Then you would want a 0.1 ms average transaction time.

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. A prevalent system will write transactions to the journal before it executes them, so it can't store the execution times with the transactions, but it could either store transaction times in a separate file or it could store the execution time of a transaction with the next transaction.

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.

External blob storage: One way to reduce the size of both transactions and the prevalent system is to save large data items outside the system if they do not need to be processed. For example, photographs are often treated as large binary objects. The clients can save them on disk and the prevalent system will just keep track of metadata about the photographs. This would not work if the prevalent system needed to process the photographs in some way, such as if it were performing face recognition. But it works well if the system is just storing a reference to the large data items.

Acknowledgements: Karl Wettin, Justin Sampson, Caleb Johnson

References

[Akka 2014] Akka persistence.
http://doc.akka.io/docs/akka/snapshot/java/persistence.html 2014.

[Evans 2005] *Domain-Driven Design* by Eric Evans, Addison-Wesley, 2005.

[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.

[Garcia-Molina 1984] *A Massive Memory Machine* by H. Garcia-Molina, R.J. Lipton and J. Valdes, IEEE Transactions on Computers, May 1984, pp. 391-399.

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

[Ongaro and Ousterhout 2014] *In Search of an Understandable Consensus Algorithm* by Diego Ongaro and John Ousterhout, 2014 USENIX Annual Technical Conference.

[Smith et al 2003] *Croquet – a collaboration system architecture* by D. A. Smith, A. Kay, A. Raab, D.P. Reed. In Proceedings of First Conference on Creating, Connecting and Collaborating through Computing, 2003, pp. 2-9.

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