Historical Modeling

Finding meaning in the relationships between partially ordered facts

Michael L Perry

Historical Modeling

Finding meaning in the relationships between partially ordered facts

Michael L Perry

Contents

Why Historical Modeling?

At the beginning of this century, a friend and I, working on building a web development business, attended a day-long conference on networking equipment and services. We saw routers designed to balance the traffic in any given network, and testing equipment designed to saturate that network and bring it down. We learned about the OSI 7-layer stack, and where different vendors and components fit in and supported that model.

This conference was eye-opening for me. But not because of all the innovative solutions that I saw before me. My eyes were opened to the absurdity of the problem itself. It seemed that we were putting far too much effort, money, time, and brainpower into solving a single problem: moving data from where it was to where it is needed.

What occurred to me is that we have invested a huge amount of effort into making sure that information flows from one computer to another. For example, it flows from a web browser on an end-user's computer to a database in a datacenter halfway around the world. Along the way it passes through network switches, load balancers, web farms, and application servers before it finally arrives.

After the data takes that long arduous journey, that same user will refresh the page and pull all of that data back again.

How absurd this all looked to me! Surely, we can find a model of computing that lets the data reside where it is most needed. If information originates from a user, why must he wait for it to cross all those layers only to be re-rendered and displayed right back to him? If he wants to collaborate with someone else, why can't they exchange just the common subset of information between themselves, rather than mixing their data together in a central database?

This observation set me on a journey to discover a model of computing that uses data where it resides, rather than relying upon the interactions between connected systems. I found hidden assumptions that we impose upon ourselves when building distributed systems: a client must wait for a response from a server in order to get the most recent data; a record can be identified by a key generated upon insertion; a URI – despite containing a host name – can identify a resource independent of its location. And the biggest assumption of them all: changes to individual resources can appear sequential, even though the aggregate changes asynchronously and in parallel.

The systems that we had always built depended upon these assumptions. And so the protocols and languages supporting those systems had to ensure that these assumptions were true. But they weren't. And so those protocols and languages were unreliable. Which leaves vendors at a networking conference selling plugs for a leaky dyke.

One-by-one I removed these assumptions until I was left with a fundamental core: a set of laws that govern information, how it interrelates, how it flows, and how it can be traversed. Then piece-by-piece I constructed a model of computing that relies only upon these fundamental laws. The

result has clearly defined behaviors, whether a system is connected or disconnected, synchronous or asynchronous. It makes no promises that it can't keep.

The model removes concepts that we used to take for granted. Concepts like changing the value of a variable over time. Concepts like locking a resource so that it cannot be changed by two actors simultaneously. Concepts like looking up the value of a resource by its ID. The challenge, then, is to build software systems under a new set of rules. Can the systems that we need be built without the assumptions that we are used to?

The answer, I discovered, is no. You still need to return to the old regime to solve specific problems. But amazingly, the larger portion of the software systems that you need can be constructed without them. These portions can be built using only the core rules of information: Historical Modeling. And when you do so, you reap the rewards of reliability, scalability, and autonomy.

In this book, I will introduce you to the rules of Historical Modeling. I will demonstrate that these rules are easily satisfiable in a distributed system. I will show how you can use Historical Modeling to build the larger portion of a software system, and how to integrate that portion with the old regime when necessary. But before I do that, I must first expose the hidden assumptions under which we have traditionally written software.

CRUD

When you look at how software is constructed today, much of it is concerned with computing the current state. Entire business applications are built on the operations Create, Read, Update, and Delete – known as the CRUD operations. In this mission to compute the current state, we have lost track of a vital piece of information: how the system came to be in this state in the first place.

CRUD destroys information about past decisions. Let's dig a little deeper into the CRUD operations to see where that information got lost. The first two operations do not destroy information.

- Create – record new information where there was none
- Read – retrieve information already recorded

Creating does not destroy any information that used to be there; it simply adds to it. Reading doesn't change the system at all. So neither of these two operations lose information.

But now comes Update. It sounds innocuous: bring some information up to date. But what happens to the information that used to be in its place? When you update a customer's shipping address, the old address is overwritten. It is destroyed. And with it, we destroy some potentially important features. Are there any orders being shipped to the old address that we need to re-route? Is there a general demographic shift that we can mine from this data?

And what was the context of that change? Who made it, and what other information was available to them at the time? This might be important for assessing data quality, and for correcting errors after they are discovered.

When performing an Update operation, all of this information is lost. Even if the system captures auditing attributes – who made the change and when – these attributes themselves will be overwritten by the next update. Updates, because they occur in situ, are destructive.

Then we come to Delete, which is destructive by its very nature. When a record is deleted, not only is the information contained within the record destroyed, but so is the audit information that might have been saved along with it. If subsequent records had referred to the deleted record, then at best those records are orphaned. At worst, the deletion cascades through the system, deleting wave upon wave of related data.

And so, while Create and Read preserve information, Update and Delete destroy it. The first step toward modeling a system historically, therefore, is to purge Update and Delete from our toolset.

Immutability

There is a word for models that don't allow Update or Delete operations: immutable. The challenge is to represent a fluid, dynamic system with nothing but immutable data structures. That challenge at first seems impossible. If the purpose of the system is to change over time, then how can we accomplish this with data structures that don't change?

The insight, however, is to recognize that the Create operation, even though it creates an immutable record, does indeed change the state of the system. If we define the immutable record thus created in just the right way, we can simulate Update and Delete operations. The difference is that, since our simulation uses only Create, no information is lost. The benefits, as we will see shortly, are tremendous.

How shall we, then, define our immutable data structures so that they simulate Update and Delete operations? It turns out that people have been doing just that for centuries. People have been recording information with immutable data structures long before computers were invented. It's just that they didn't call them "data structures"; they called them "documents". Historical Modeling simply recalls, and in some cases formalizes, these old techniques.

Let's look at a common example: accounting. Business is transacted not in the writing and rewriting of the current balance, but in the accretion of historical records.

Accountants practice the process of writing to a ledger all of the transfers of value from one account to another. At any point, the values of the transfers can be added together to obtain the current balance. But this balance is secondary. The important information is in the historical ledger of transfers themselves. It's in the history. These records capture the context of each transaction: who was involved, what was exchanged, and for what purpose. They record the knowledge that was present at the time, and upon which the business decision was made. They capture the transient relationships among the parties doing business with one another.

This information is valuable apart from its role in helping us determine current state. It helps us to reconcile different versions of history. It helps us to audit the behaviors of the past. It helps us to correct errors, not by changing history, but by compensating. And it helps record the resolutions of disputes, so that they are not raised again in the future.

Accountants' pencils have no erasers.

We understand the value of history when dealing with money. Why then do we build software on databases that overwrite history in an effort to maintain the current state?

SOAP and RPC

The year after the networking conference, I joined a team working on a gift card system for use in restaurants. It was an exciting and challenging project. At the time, most businesses were only connected to the Internet over a slow and unreliable dial-up connection. Restaurants in larger cities sometimes had DSL, but even those networks were not always connected. A key selling point of this new Gift Card system would be that it would continue to work when the network was out. To accomplish this, we chose to cache card balances at the restaurant, and exchange messages when the network was available.

For our message exchange, we decided to use a new protocol called "SOAP". SOAP, if you don't recall, stood for the Simple Object Access Protocol. One of my teammates pointed out that it was not Simple, it was not used to Access Objects, and it was not (at least at the time) a clearly specified Protocol. We were building the client side in C++, and the server side in Java, but the standard was still several years away from making such interoperability simple.

Nevertheless, we set ourselves the challenge of using SOAP to exchange data between clients in the restaurant and servers in our datacenter. We all read a book (Understanding SOAP, by Kenn Scribner, Mark C. Stiver) that taught SOAP while building a library that generates C++ function calls from XML schemas. The code in this book actually generated machine code on-the-fly to extract parameters from a stack frame, generate a SOAP call, and then turn the resulting XML back into a stack frame containing the binary structure of the result. There were several reasons we discovered that this was a poor choice for the basis of our application message interchange. But the biggest one was that this put us in the RPC mindset.

RPC, or Remote Procedure Call, is the practice of turning a procedure invocation complete with its parameters into a network request. The call blocks until the response is received, and then the response is converted into the return value. The book on which we based our implementation did this literally at the binary level.

Because we focused on the library included in this book, we only considered the RPC capabilities of SOAP. But SOAP also had another mode of operation. It could be used as a document interchange instead of remote procedure calls. If we had focused on this style of use, perhaps things would have turned out better.

Works on localhost

Thinking, as we were, in terms of remote procedures, we came up with a set of function signatures that solved our particular problem. That was to bring down a copy of gift card values to the

restaurant, and to send up a set of financial transactions. These transactions were Add Value, Redeem, and Request Seed. Request Seed was a special transaction that told the server to send a fresh set of card values to the restaurant.

When we started, things seemed to be going well. But as we continued building out this system, we noticed more and more ways in which remote messages were not like procedure calls. At first, we got interoperability between C++ and Java working, at least for the limited set of SOAP messages that we were exchanging. Everything appeared to work just fine in development. But then we deployed to test.

The test machines were not running both the client and server on the same box. They were configured to run the server on one machine, and clients on several others. Once the system was in the test environment, we started to lose transactions.

One of the tests involved running a batch of automated transactions from the various clients and comparing card balances at the end. These tests ran overnight, and in the morning we observed the card balances. Some cards still retained their higher original balance on the server, despite being redeemed on one of the clients.

We diagnosed the problem and determined that transactions were being removed from the client's outgoing queue before being sent to the server. If there was an intermittent network failure, then the client would retry sending the transaction a few times before giving up. It would log the error, but the transaction was lost.

This was a simple fix. Just keep the transaction in queue until the client heard back from the server. Once the client knew that the server had received the transaction, it could remove it from its queue.

We made the change, deployed to the test environment, and again ran it overnight. In the morning, several of the cards had lower balances on the server than on the client. We had the opposite problem!

We tried to reproduce the issue in development, but the balances always turned out to be correct. We coined a phrase to express our frustration: "Works on localhost".

The problem, we finally deduced, was that a request was making it from the client to the server, but its response was not making it back. The server was decrementing the card's value, but the client was not removing the transaction from the queue. The client would continue to re-send the transaction until the network connection was restored, at which point the effect would be duplicated and the balances would disagree.

This problem arose because we were thinking of network requests and responses as procedure calls. Could a procedure ever fail to return to the caller? It would be as if the procedure's operations completed successfully, only for the "return" statement to thrown an exception. This failure condition never happens in procedure calls. RPC tricked us into thinking that it wouldn't happen during communications either.

The Two Generals Problem

I would later learn about the Two Generals Problem. This is a story that illustrates a fundamental truth about distributed computing. The story begins with two generals of cooperating armies whose

mission it is to capture a besieged city. They must both attack on the same day. If only one attacks, they will certainly loose. They are also separated by hostile territory. The generals themselves must not cross the gap to coordinate with one another, as they would be captured. So they must send messengers through enemy territory to agree upon a day for the attack. Any messenger might be lost, but the generals must together choose a day, and both must know for certain that the other agrees with that choice.

What then is the protocol? What messages and responses can the generals send through enemy territory in order to be sure that they reach a mutually understood agreement?

If one general sends a messenger to the other with the message to attack tomorrow, how will he be sure that the message will be delivered successfully? Perhaps he should send two. Will one of them make it? Send ten? The probability increases, but this isn't a question of chance; it's a question of certainty. If the second general never receives the message and the first attacks alone, then all will be lost.

So perhaps the protocol includes confirmation. The second general sends back a message confirming that the plan to attack was received. Now the first general only attacks if he receives confirmation.

But how can the second general be sure that the confirmation made it back to the first? If it doesn't, the first will not attack. If the second general attacks alone, then all is lost.

So perhaps the protocol includes acknowledgement. The first general sends back a message acknowledging that confirmation was received. The second general can then attack with confidence, but only if he receives acknowledgement.

You can see where this is going: a stalemate of indecision. At some point, no matter what protocol you choose, there is a final message that influences the recipient's behavior. If that message is received, the recipient will act one way; if not, he will act another way. There may be later messages in this protocol, but this is the last one that influences behavior.

Given that this is the last message that influences behavior, what is the sender's plan? The sender does not know whether the message will arrive. Nevertheless, he must already have determined his own behavior. So based on whether this particular message arrives or not, the sender and the recipient may or may not agree.

Thus, there is no solution to the Two Generals Problem. There is no message passing protocol that can guarantee that two parties reach an agreement, such that both know that agreement has been reached. And this was precisely the problem that we the Gift Card team believed we needed to solve. No wonder we were frustrated!

The decision maker

Even though we didn't know about the Two Generals Problem at the time, we believed that we had to solve that problem to complete a gift card transaction. The client had to know that there were sufficient funds on the card to complete the transaction. The server had to agree that the transaction

had taken place, and deduct the balance accordingly. Either the transaction took place on both sides, or it took place on neither. Both sides had to reach a mutually understood agreement.

But in fact, that problem statement is overconstrained. Yes, indeed, both sides need to reach an agreement. However, they do not need to both understand that agreement had been reached. By relaxing this constraint, we turned the problem from an impossible Two Generals into something that we could solve.

The key to solving this problem was in appointing a decision maker. In our case, the client was ultimately responsible. The client would do its best to determine the available balance, but no matter what information the server might have, the client would make the final go/no-go decision. Once the decision was made, the server would eventually have to agree with it. But the client did not need to know that agreement had been reached.

I should point out that our business customer was not pleased with this result. The server was ultimately aware of the balance of each card. Yet we, the development team, were arguing that the client would have the final say. If the client decided that the card should be redeemed, even if the server knew that it had insufficient funds, the transaction could not be rejected. How could it be any other way? If the server made the ultimate decision, then there would be no performance or resiliency benefit gained by caching card balances on the client. A key selling point of our solution was that it worked while disconnected, and so this was the only choice permitted us. Eventually, our business customer understood that this must be the case.

Once we had a decision maker, the Two Generals Problem disappeared. No longer must both sides understand that an agreement had been made before acting. One decides, and then continues the conversation until he knows that the other is aware of that decision.

Idempotence

Having reframed the problem to make it solvable, we set forth implementing the solution. Remember that we faced first the problem of lost transactions, and then the problem of duplicated transactions. We had to be sure that a transaction would not be duplicated. The word for that, I would later learn, is “idempotence”.

Idempotence is expressed informally as the property of a system that it will only apply a message once, even if it receives it multiple times. Formally, it is expressed as:

```
1  f(x) = f(f(x))
```

The function `f` is the behavior of the system as it receives a particular message. You could say that the function **is** the message, or at least the effect of the message on the state of the recipient.

The variable `x` is the state of the recipient before the message is received. And so `f(x)` is the state of the recipient afterward. The equation above simply says that a message is idempotent if the effect that it has on a recipient that already received the message is … well, nothing. The state of the recipient is left unchanged.

Let's take a look at some messages to see if they are idempotent. For example, suppose that the message `f` is a sale: that Bob Jones just bought a latte for $5. The variable `x` is the state of the system before the message was received. This state captures the balance of each person's gift card.

```
x = {
  Sally Jane: $15,
  Bob Jones: $12,
  James Tan: $7
}
```

The value of `f(x)` would then be the state of the recipient after applying the message. Bob Jones' balance is decreased by $5, but everyone else's remains the same:

```
f(x) = {
  Sally Jane: $15,
  Bob Jones: $7,
  James Tan: $7
}
```

Now if we receive the message a second time, we apply it again:

```
f(f(x)) = {
  Sally Jane: $15,
  Bob Jones: $2,
  James Tan: $7
}
```

This message is not idempotent, because `f(x) ≠ f(f(x))`. And Bob is not happy about that.

In this example, the message `f` is a sale, and the state `x` is the set of card balances. This choice of message and state leads to a system that is not idempotent.

Idempotence is important in a distributed system, because the sender cannot know whether a message was received. So it must have the option of sending it again. Only if the message was idempotent can it know that resending it is safe.

First move it, then process it

In the above example, because the state was the set of card balances, receipt of the message was the same as processing the message. The function f modified the state to calculate the new balance. Our overnight experiments on the Gift Card project showed us that this was a problem. We computed

the balance of the card upon receipt of the message, and in doing so lost valuable information. We had to preserve that information to make our messages idempotent.

Our first step was to separate transmission of a message from its processing. After doing so, the state of the system was no longer about the result of the transactions. It was only about the receipt of those transactions. For example, the initial state of the server might be the following:

```
x = {
  MessagesReceived: [ 'abc', '123' ]
}
```

Now we are no longer talking about the balance of cards. We are only talking about the IDs of messages received. These IDs, in our case, were GUIDs generated in the restaurant by the originating client. Today I would make a different decision, but we'll get to that later. For now, let's continue with the GUIDs.

Suppose the client gives Bob's purchase the GUID '456'. It sends the transaction in a message which we will call `s` for "send".

```
s(x) = {
  MessagesReceived: [ 'abc', '123', '456' ]
}
```

Notice here that the recipient's state has changed to include the GUID of the sent transaction. If the sender is unsure whether the recipient received it, then it can retry the message. The result on the recipient's site would be:

```
s(s(x)) = {
  MessagesReceived: [ 'abc', '123', '456' ]
}
```

Notice that `s(s(x)) = s(x)`. The collection of messages received did not change, because the GUID was already present. In other words, the "send" message is idempotent. The processing of this transaction – reducing Bob's balance – can happen later.

The sender at this point has no idea whether the message was received, so it must continue to try indefinitely. That is, unless the recipient confirms receipt of the transaction. Let's make it do so in the form of a return message. For all the same reasons that we've already discussed, the confirm message must also be idempotent. We'll therefore take a look at the sender's state before this conversation started.

```
y = {
  MessagesToSend: [ '456' ]
}
```

This is the sender's outgoing queue. The sender periodically checks the queue and sends (or resends) the transactions it finds there. When the recipient receives that transaction, it responds with a confirmation message, `c`. The sender computes its state after confirmation:

```
c(y) = {
  MessagesToSend: [ ]
}
```

Just as with any message, the confirmation can be duplicated. If it is, the sender simply ignores it. Thus the state after receiving confirmation twice is:

```
c(c(y)) = {
  MessagesToSend: [ ]
}
```

Since `c(y) = c(c(y))`, the confirmation message is idempotent. The recipient can confirm repeatedly without causing any change on the sender. The protocol we chose here is simple: the recipient responds with a confirmation any time a transaction is sent, even if it had already been received. The recipient assumes that the sender will stop once it receives confirmation.

At this point, the two parties have reached an agreement. They both know that the transaction has been sent. However, the recipient doesn't know that agreement has been reached. This is what makes this problem different from the Two Generals Problem. The recipient doesn't need to know.

The processing of transactions was not idempotent. This lead to lost and duplicated transactions. But by separating the transmission of a transaction from its processing, we can impose idempotence where there was none, and fix those problems.

Technical benefits

Given the requirements of immutability and idempotence, it seems that it would be difficult to construct working systems based upon these restrictions. However, as we have already discovered while exploring accounting records, these restrictions provide many business advantages. These advantages have caused our ancestors to gravitate toward systems of record keeping based on historical documents. The advantages of historical modeling to business visibility cannot be overstated. If these were the only benefits, they would be enough to motivate us to continue building systems this way. However, there are even more gains to be had in the realm of technology.

It is rare for a truly useful piece of software to run on only one computer. Web applications run at least on the user's computer – in the browser – as well as in the server. More often, the server is not just one machine, but a cluster of web servers, application servers, cache servers, and database servers.

Mobile applications, too, don't run just on the phone or tablet. They also run partly in the cloud. Messages are pushed out to the machines belonging to network providers, so that push notifications can be sent back down to the device. If the application is collaborative in nature, then it runs not just on one user's device, but also on those of her collaborators.

When multiple machines work together to run an application, each one will have a different subset of the information at different times. No single machine will ever know the entire state of the system. We therefore cannot program a system based on state if we are to expect to reason about its behavior as a whole. Instead, we must accept the fact that decisions are made elsewhere without our knowledge. Once we learn about those decisions, we must be able to incorporate them into the subset of facts that we know. At that point, we can interpret the new information and arrive at a new consistent view of the world.

That is precisely what Historical Modeling does for us. It gives us a set of rules for capturing decisions where they are made, for moving those decisions where they are needed, and for understanding what they mean once they get there.

Autonomy

Distributed systems constructed with historical models are made up of nodes that can act autonomously, yet reach consistency. They each express precisely what subset of information they require. When a decision needs to be made, these nodes act upon what they know at that time. They don't wait for other nodes to give them more information. They don't block the transaction in order to ask each other questions. They decide, record their decisions as historical facts, and move on. The benefits to performance, scalability, and efficiency are measurable.

The Gift Card system was just one example. The client within the restaurant kept enough information to make decisions autonomously. The server in the datacenter honored that decision, even if it later discovered that the card had insufficient funds. Yes, this introduced risk in the form of unrecoverable balances, but it more than made up for that risk by allowing the system to work while the network was down.

Mobile applications are another example. If they are built using a historical model, they will continue to serve the user even when a network connection is unavailable. They have the subset of information required by that user. There is no need to pause and contact a server to get more. When the user makes a decision, that fact is added to the local subset of information immediately, and has the effect that the user desires. A round trip to the server is not required. As a result, the mobile app is fast and responsive, even if the connection is slow or broken.

Collaborative applications using a historical model will detect and reveal conflicts. They will capture enough information to recognize not only that a conflict has occurred, but also what information was

known to each party when they made their conflicting decisions. Based on this information, conflict resolution can either be automated or manual. The conflict and all possible candidate resolutions can be shown to the interested parties so that they can resolve it in their own way. No matter what the strategy, the resolution of the conflict will be recorded as another fact, becoming part of the historical record understood by all parties.

Git as a historical system

Given these technical benefits, it is not surprising that many systems have already been modeled historically, even if they don't use that nomenclature. The distributed version control system Git, for example, captures changes to source code as historical facts called "commits". A commit is a set of changes made to a project by a single developer at a single point in time. Commits are stored in a local repository, so that the server does not need to be consulted for most operations. The developer clones a repository, collecting a subset of commits to work with. The current state of the source code can be constructed from the commits without reconnecting to the remote host.

The developer works disconnected from the server to construct a new set of commits on his own. While he works, he is not connecting to the remote host to lock files or check for the most recent changes. He is working in complete isolation; no round trip to the server is required.

When a conflict occurs, as it must in any distributed system, the developer finds within his local repository all of the information necessary to resolve it. He has the identification of the collaborators (possibly even himself) involved in the conflict. He knows exactly the context of the change – what the code looked like at the time it was modified. And he even has from the commit comments some clue as to the intent of the programmer.

Based on all of this information, the developer can resolve the conflict himself. He doesn't need to involve the server. In fact, because of the nature of Git branches, he can choose to let the conflict stand as long as he pleases. There is no immediate need to for the conflict to be resolved before work can continue. But when a resolution is made, it is recorded as another commit. That commit becomes part of the history so that all parties involved can see that the conflict has been resolved, and understand the effect of this resolution.

The Rules of Historical Modeling

I have uncovered some of the assumptions under which the software industry has been constructing systems for several decades. One assumption states that there is one true state of an entity, and that that state can be Created, Read, Updated, and Deleted. Another asserts that procedure calls can be made across machine boundaries. Another says that the system of record must be the deciding party.

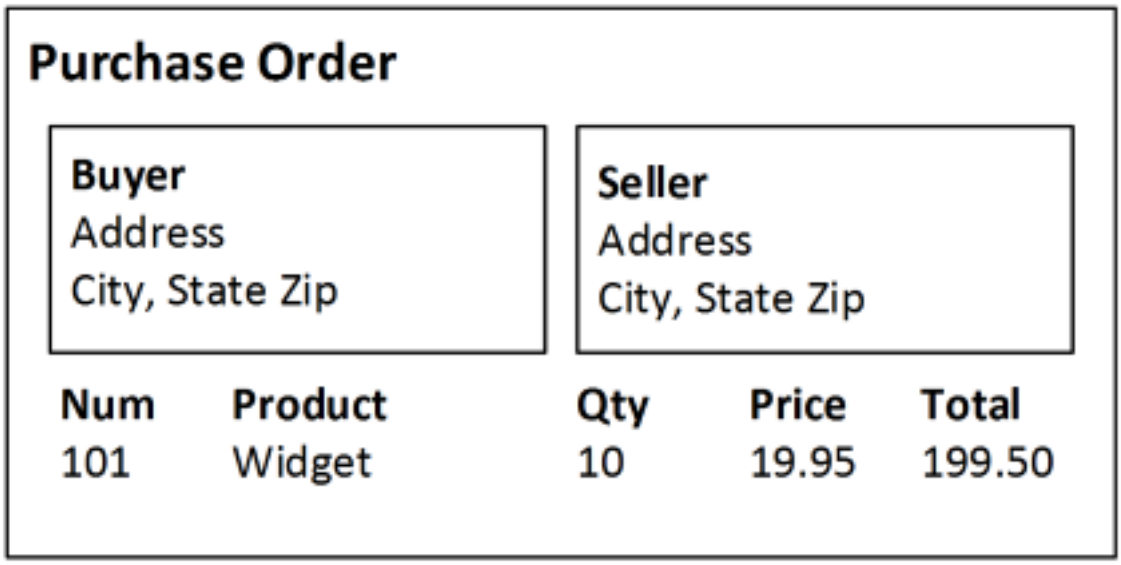
These assumptions have crept into the very tools that we use to build software. In an object-oriented language, the property of an object has a single value. It does not represent the multitude of possible values that the property might have on different machines. In a relational database, UPDATE and DELETE are first-class commands that permit destruction of information. In HTTP, requests block until the response is available, mimicking the behavior of a procedure call. In REST, a URI identifies not only the resource, but also names the system of record that must be contacted to get its current representation.

As I uncovered these assumptions, I revealed the problems that they cause. When we use tools that are built on top of these assumptions, we will have a difficult time avoiding these problems. Instead, I propose that we construct a new set of assumptions, and from those derive a new set of tools. This chapter lays out those assumptions, and defines them as the rules of Historical Modeling.

The nature of historical facts

Let's go back in time to a world before computers. How was business transacted in this world? Rather than updating the current state of the world in a large database, information was recorded and shared in the form of documents.

Suppose a customer places an order for ten widgets. This decision is captured as a purchase order. The purchase order references the two parties: the buyer and the seller. It also references the product – widgets – by the catalog number assigned by the seller.

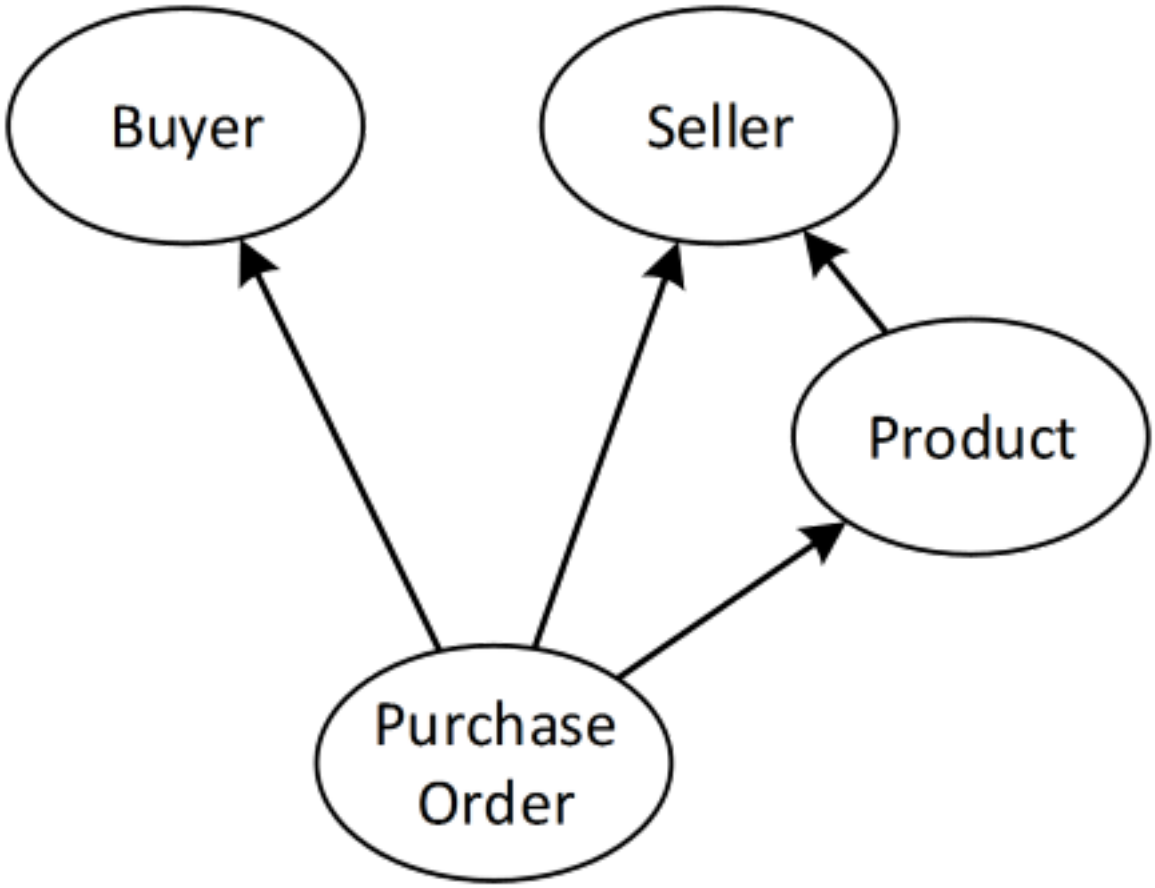


Purchase Order

The purchase order is a **fact**. It is a historical document that records a decision. It is immutable: neither party can change the purchase order itself. They can only amend this document with another one.

The purchase order fact refers to a few other facts that came before. It refers to the buyer and the seller as distinct legal entities. These entities were created with their own set of historical facts – documents that were filed with their respective regulating bodies as articles of incorporation. These facts were created well in advance of the purchase order.

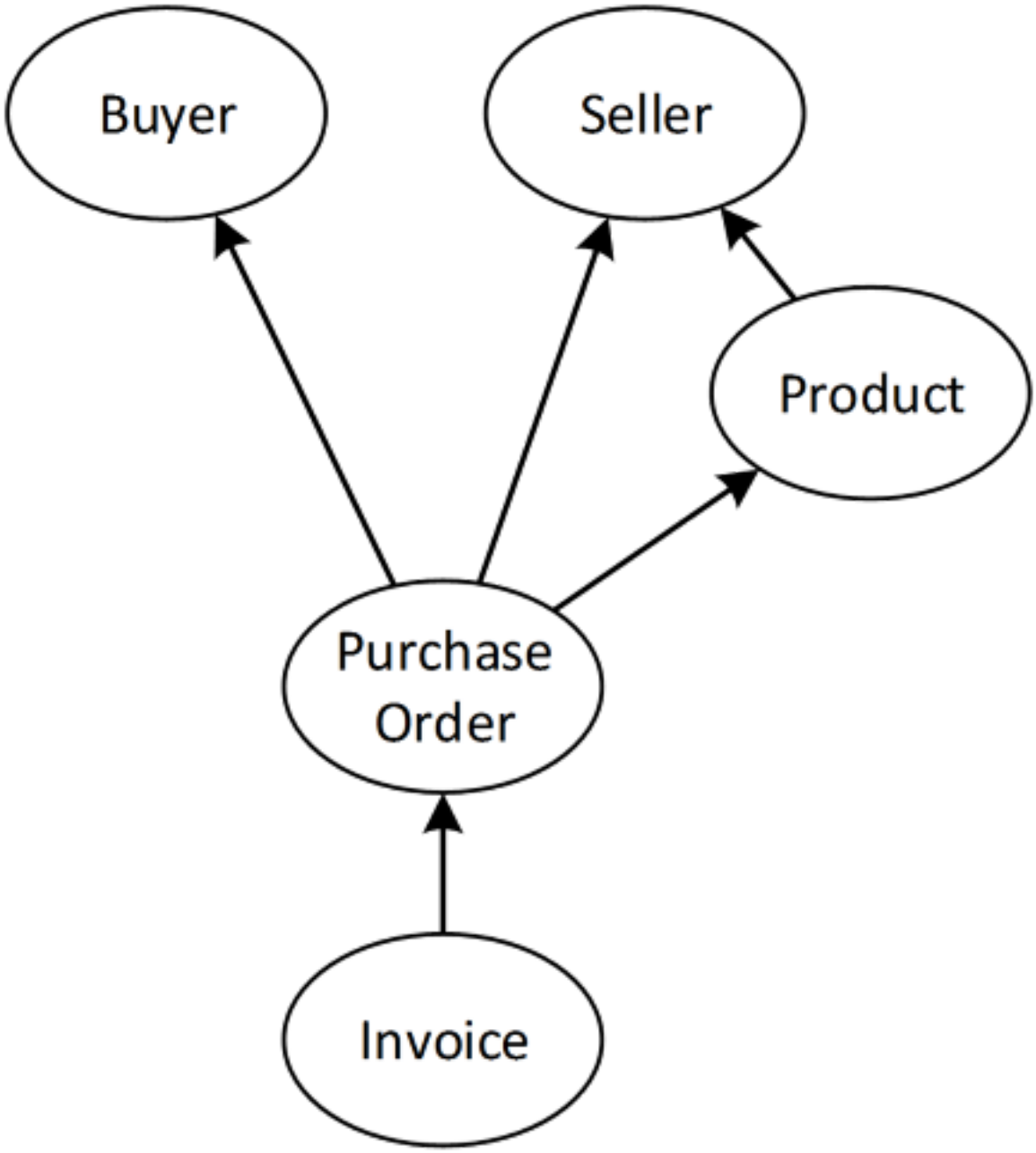
The purchase order fact also refers to the catalog number. This is a historical fact that the widget was published in a catalog of available products with a listed price. The catalog, once published, is not changed. It is only amended by publishing subsequent catalogs adding and removing products, and changing their prices.



Purchase Order Fact

All of these facts that came before the purchase order – the buyer, the seller, the product – are **predecessors**. A predecessor is a fact that necessarily came before. A fact cannot refer to another fact that came after; that would only be possible if facts could be altered. At the time the fact was created, the successor did not exist.

Let's continue the story of the buyer and the seller. The seller receives a copy of the purchase order, and then sends the buyer an invoice. The invoice is another historical fact. The predecessor of this fact is the purchase order.



Invoice Fact

The inverse relationship to predecessor is **successor**. The invoice is a successor of the purchase order. Facts don't directly know their successors. They cannot, since they are immutable. But the system can be queried to find the successors of a given fact. This is how a system made up of immutable facts can appear to change. Successors accrue over time, and thereby change the results of a query.

Both the buyer and the seller in this story have their own view of the world, based on a **subset** of history. At any given time, these world views may differ. The buyer knows about the purchase order before the seller does. And the seller knows about the invoice before the buyer. There is no

single state of the system, and no authoritative source of truth. Nevertheless, every party eventually learns of all facts in the system. They understand the predecessor/successor relationships among these facts. Each party examines the subset of information available to them at the time to draw their own conclusion about current state, and the transaction eventually draws to completion.

The nature of historical facts, in summary, is that:

- Facts are **immutable**
- Facts refer to others that have come before: their **predecessors**
- The system can be queried to find the **successors** of a given fact
- Only a **subset** of history can be known at any given time

Remember that our story of buying and selling takes place before the creation of computers. Historical facts have behaved according to these rules since the dawn of recorded history – literally.

The rules of historical models

The nature of historical documents leads us to a specific set of rules. By this, I don't mean suggestions or best practices. These are strict formal rules that follow directly from the nature of facts. We start with the goals, and from there derive the rules that must be true if those goals are to be met.

We'll define the goals in terms of **facts** (units of information) and **nodes** (individual actors who observe the facts). A node is typically a single machine in a computer network. A node observes – or knows about – certain facts within the system. The decisions that it makes are deterministic, and based only upon the facts it observes and input that it receives directly. The goals are these:

- Information is never destroyed
- Nodes behave independently
- Each node observes only a subset of all facts
- Nodes tend toward consistency

Beginning with the first goal, then, we derive the first rule. If a fact is the unit of information, and information is never destroyed, then it follows that…

> Facts are immutable

A fact records an action or a decision that was made at any given node at some point in time. Once that decision is made and recorded, it is forever true. The fact of its historical existence cannot be changed or deleted.

Now that doesn't mean that we cannot change our minds. We are certainly free to make a new decision. When we do, this new decision is recorded as a new fact. The old fact still exists, but it

now co-exists with a new fact. We must find a way to reconcile a system in which both facts exist, and to understand what it means.

The immutability of facts must somehow not impede progress. It must not conflict with fourth goal: the tendency toward consistency. That is how meaning evolves as the system progresses from one historical fact to the next. To accomplish that, we must honor the second rule.

> A fact refers to its predecessors

To understand the meaning of a system in which two contradictory decisions co-exist, we must understand which one came first. Did one decision supersede the other, or did they arise independently? If one fact has knowledge of another, then we can conclude that it represents a modification of or amendment to the first. But if neither fact knows about the other, then we recognize that they exist in conflict, requiring a later decision.

The predecessor relationship captures all of this information. If one fact refers to another, and if those facts can't be changed, then the fact must follow the one to which it refers. It must have occurred later in time. The predecessor preceded its successor in the very literal sense of the word: it happened earlier in time.

> Predecessors occurred before successors

The predecessor relationship also captures the knowledge of the prior fact. The second decision was made with the knowledge of the first. The first couldn't have known about the second; it hadn't happened yet. But the second clearly knew about the first. We know that the predecessor wasn't added after-the-fact, because facts are immutable.

The presence of a successor changes the interpretation of its predecessor. The existence of the first fact is not in dispute. The content of the first fact is unchanged. The first fact is not destroyed. But its meaning changes in light of its successor. Based on how successors are used, this new meaning can augment, replace, or even nullify the original intent.

Partial order

If the rules outlined above were the only conclusions we drew, then we could model a historical system in a number of ways. For example, we could line all of the facts up in the order in which they occurred, and let each one refer back to its immediate predecessor. This stream of events would form a chain stretching back to the beginning of time. However, such a model would not permit us to satisfy all of our goals. Specifically, this choice would not allow nodes to behave independently, or to observe only a subset of facts.

For nodes to behave independently, they need to be authorized to make and record decisions without consulting with their neighbors. This must inevitably lead to the existence of two or more facts that do not refer to one another as predecessors. If neither of two facts refers to the other, then we cannot clearly say which of the two came first. We cannot impose a total ordering of facts.

Facts allow a partial order

The terms "total ordering" and "partial ordering" come from mathematics. A set is totally ordered if you can identify, for any given pair of elements, which one came before the other. A totally ordered set is like the counting numbers. You can clearly enumerate the order in which they must occur.

A partial order, however, allows some ambiguity. There are some pairs for which the before/after relationship is not known. When it is not clearly stated, then either order must be allowed. A partially ordered set is like a genealogy. I know that I am the child of – and therefore came after – my father. But if I compare myself with someone with whom I share no family relationship, then the order is unclear. I speak here only of parent/child relationships, not birth order. So by this definition, I cannot even say whether I came before or after my brother (even though he is older).

The "comes before" part of the predecessor relationship is called **causality**. A predecessor is causally related to its successor. Two facts that are not causally related are called concurrent. These definitions are a bit different from common vernacular. A predecessor does not necessarily "cause" the successor in any real way. Nor can we say that two unrelated facts indeed happened "concurrently". However, these are the accepted definitions given by Leslie Lamport (Time, Clocks and the Ordering of Events in a Distributed System), and so we continue to apply them.

We borrow many truths about historical facts from the mathematical definition of partial order. One that plays particular importance is this.

Causality is transitive

If one fact is the predecessor of a second, and the second is a predecessor of a third, then the first comes before the third. We don't need to explicitly list the first as a predecessor of the third. The fact that it came before is inferred by the transitive nature of causality.

So given any two facts, I might be able to follow a chain of predecessor relationships from one to the other through some number of intermediates. But then again, I might not. These two facts might have occurred independently on different nodes. These two nodes have the authority to make and record decisions without consulting one another. The model must allow a partial order if nodes are to be permitted to act independently.

Collaboration

Let's further examine the goal of nodes tending toward consistency. Nodes will use facts as a means of communication. They share facts with one another. Once a node learns of a fact, then it observes it. It brings it into its model of the world, upon which it will base all future decisions.

For nodes to use facts as a means of communication, each node must recognize when a fact learned from a neighbor is actually one that it already observes. The collaborators need to identify facts, and know when they are indeed the same. That identity must be deterministic. If two nodes exchange

two facts, they must both agree whether they are indeed the same. If a third node enters the conversation, it must make the same determination. The only way to guarantee determinism in a system of independent nodes is to introduce the following rule.

> A fact is identified by its content

The rule is two-fold. First, if two nodes observe two facts that have exactly the same contents, then they are both actually observing the same fact. And secondly, there is no subset of a fact's contents – and no identifying attribute apart from its contents – that can be taken as an identifier. The identity of a fact is its content, its whole content, and nothing but its content.

The need for this rule comes from the goal that nodes behave independently. When a node makes a decision, it must be able to record that decision as a fact without the need to consult any other node. It should not need to allocate a unique identifier by consensus, pre-arrangement, or central authority. Using the content as the identity is the only deterministic way to guarantee uniqueness.

It should be stated that the content of a fact includes the identities of its predecessors. Two facts that are similar in all respects, except that they refer to different predecessors, are indeed discrete facts. This observation is not to be taken lightly, however. Remember that the identify of a fact is the same as its content. Therefore, in a very real sense…

> A fact contains its predecessors

If a fact knows the identity of its predecessors, and that identity is their content – including their predecessors – then a fact actually contains its entire lineage. This implies that not only is causality transitive, but so is observation.

> If a node observes a fact, then it also observes that fact's predecessors

This set of rules governing the identity of facts is peculiar to Historical Modeling. Most other systems define an identity separate from – or as a subset of – the content of a record. Relational databases, for example, assign a primary key to the rows of a table. The key is only part of the content of the row. Other columns contain data related to that key. Document databases, as another example, typically define an ID separate from the content of a document. One useful function of these systems is to fetch a record by its key or ID. An actor can know the key or ID, but still not know the content of the record.

In a Historical Model, however, an actor that knows the identity of a fact already knows its content. It therefore makes no sense for such a system to offer "fetch" as a function. The identity also includes the identities of the predecessors, so "find predecessors" is not a useful function either. Anyone capable of asking the question already knows the answer. The only function that makes sense in a historical system is "find successors". That is a function that we will make much use of in the remainder of this book.

We have derived the rules of Historical Modeling from the goals. The goals are to model a system in which no information is destroyed, nodes act independently, observe a subset of facts, and approach consistency. These goals lead directly to a set of rules. Facts are immutable. They refer to their predecessors, with which they define a partial order. The content of a fact is also its identity, which includes the identities (and therefore contents) of its predecessors. The only useful operation is therefore to query for the successors of a given fact.

I have shown that these rules are necessary given the goals of Historical Modeling. I hope to also convince you by the end of the book that they are sufficient.

Timeliness

In a system based on the exchange of historical documents, not all parties will know about all facts at the same time. This is one of the greatest strengths of historical modeling, but also one of its important limitations. It is impossible to reject a fact based on the time at which you learn of it. The reason is that other parties will have learned about it earlier, and would therefore have come to a different conclusion about the fact. For every party in the system to eventually reach the same conclusion, that conclusion cannot be based on timeliness.

This causes significant problems in systems that do not recognize this limitation. Several legal documents, such as tax forms, checks, and invoices, have explicit due dates or expiration dates. If the form is received after the required date, then it will not be honored. The sender must go to great lengths to prove that the document was written and transmitted on time, or suffer the consequences of a failed transaction. In such situations, the sender believes one thing – that he met the deadline – while the recipient believes something else. Only by arbitration of a central authority can these situations be resolved.

To design a system that does not rely upon a central authority, we must respect that documents will be received late. In a truly historical model, a fact is not rejected based on the time at which it was received. At best, we can record the fact that a fine was levied or an opportunity was lost due to the failure of information to arrive at a certain place by a certain time. But we cannot prove that the information did not exist somewhere else at that time. And when the fact arrives later, we must decide how we are to react to it. All parties must honor the existence of the facts, no matter when they learned about them, and draw the same conclusion. Perhaps that conclusion is that the sender still owes a fine. But timeliness alone did not determine that outcome.

Such are the rules of a historical model. They follow logically from the desire to capture the full history of a system with several parties, separated by time and space, exchanging historical facts via documents. Those documents must be immutable. Two documents having the same content represent the same historical fact. We cannot guarantee – and therefore cannot rely upon – there being only one successor for any given fact. And we cannot change our interpretation of history based on the timeliness of our knowledge of it.

Limitations of historical models

With the power of historical modeling comes some constraints. These constraints make it inappropriate to apply historical modeling to certain types of systems. In these situations, it is best to model all or part of the system statically – that is, using a method that captures current state – and integrate where appropriate. Fortunately, good integration strategies are available.

No central authority

A historical model allows for decisions to be made with autonomy. Each decision is recorded in the local history, and eventually shared with the rest of the system. As a result, the system cannot reject facts based on age or current state.

Decisions that were made in the past are approved locally, with only the information available at the time. No remote part of the system needs to be consulted. That decision cannot be rejected post-facto.

This makes historical modeling inappropriate for parts of a system that require a central authority. For example, a conference room reservation system will need to know with certainty whether a room was available at a certain time. When a reservation is approved, the approver needs to know that no other reservation for the same room at the same time has been approved. That decision must be made by a central authority.

A historical model may be applied around the edges of a central authority, so long as that central authority itself is using a static model. The historical model can capture the fact that a request has been made. This occurs at the point of request, such as at a user's workstation or a device mounted by the door, and these facts find their way to a central authority. The historical model can also capture the fact that a request was approved. This occurs at the central authority and moves out to the devices at the edge. But a historical model alone cannot say for certain whether a room is available at any given time. That would require that the model know that a reservation has not been approved, which is impossible given a subset of history.

To solve the problem, the system should include a central authority with a static model. The historical model records the reservation requests and approvals, but the static model determines availability. The central authority need not be a single machine; it could be a cluster of machines. As long as the members of this cluster have access to the same static model, they can act with singular authority. The shared static model needs a locking mechanism to help this cluster coordinate their actions. Relational databases support transactions, which answer this need well.

The central authority will then record the approval or rejection of the request as a successive fact. This fact will find its way back to the client from which the request came. The historical model provides all of the benefits previously mentioned: a complete history of the request, an eventually consistent view of current state, and a mergable communication mechanism. The one component that is better modelled statically is the one that requires central authority: room availability.

No real-time clock

A time-sensitive request must be fulfilled within a specified period of time. If it is not, the request is invalid. Such requests are common in real-time systems such as factory automation. A request for a door to open or a robotic arm to move must be fulfilled within a narrow span of time. If the message does not arrive in time, then the request must be rejected.

Facts in a historical model, however, are honored no matter what the time frame. The decision is made at the time that the fact is recorded, and cannot be rejected thereafter. It may take an indeterminate period of time to transmit the fact. The recipient is simply informed of something that has already happened in history.

While it might be appropriate to model the input or output of a real-time factory automation system historically, the software that runs the factory itself should use a real-time model. These models are specifically designed to provide time-sensitive fail-safe behavior. If a message fails to arrive at the right time, the system defaults to safe operation. And once the message does arrive, late as it is, the system ignores it so as not to cause any damage.

No uniqueness constraints

In a historical model, any query for successors of a fact might return multiple results. It is not possible to constrain a query to return only one result. The consequence of this is that a domain that requires at most one result cannot effectively be modelled historically.

For example, a login that requires a unique user name should be supported by a static model. A historical model would be unable to enforce the uniqueness of a user name.

At best, a historical model might include a fact containing only the user name. Because a fact is uniquely identified by its value, there is logically only one fact with this exact user name. However, the fact could contain nothing that could differ from one user to the next. If it contained information in addition to the user name, then two or more facts could again exist with the same name. They would no longer be unique.

To correlate a distinct user name with a user, therefore, would require a successor fact. Identifying the user for a given user name would require a query for the successors of the user name fact. Such a query cannot be guaranteed to return at most one fact. The possibility always exists for it to return more.

To model a system that requires uniqueness constraints, you must use a static model. The model can be consulted to determine if the desired value is already in use. The indexing and transactional features of a relational database once more come into play.

That static model must also be centrally located. A replica of a static model cannot enforce uniqueness. An insertion into one copy would need to block in order to consult the others. Only if that unique value is not reserved in a quorum (usually a simple majority) of replicas can it be accepted. A consensus algorithm such as Paxos can be employed to reach a quorum.

If uniqueness is required, such as registering for a user name, a historical model could be used for registration requests, as well as for acceptance or rejection responses. The requests can be recorded as facts by clients at the edge of the system. These facts will make their way to a central authority that has access to a static model. The static model enforces uniqueness constraints. That central authority will decide whether to approve or reject the request based on the static model, and then record that decision as a successor fact.

The response will find its way back to the client from which the request came; only then will the client know whether the requested user name is unique. They will query for successors to the request fact – the acceptance or rejection. Once they have one successor, they will know the answer to the uniqueness question. However, there is no guarantee in the historical model itself that the request will have no further successors! That assurance comes only from the trust that a central authority is making the decision, with the help of a static model that can enforce uniqueness.

No aggregation

After a certain amount of activity, a system might be expected to provide an aggregate or summary of that period's activity. For example, a financial ledger could be closed at the end of a day, a month, or a quarter. The system would then produce a summary that records the total of that period's transactions. From that point forward, no additional transactions would be allowed into that period.

A historical model cannot guarantee that all facts within a given period have been seen. The system responsible for generating the aggregate might not have all of the period's records at the required time. If it receives a fact after computing and recording the summary, then it is not permitted – by the rules of historical modeling – to reject it. The decision was made elsewhere, and the fact of that decision was merely shared.

Three strategies exist for dealing with aggregation of historical facts: central ledgers, map-reduce, and block chains. A central ledger is by far the simplest of the three. A central ledger uses a static model to tally which facts have been included in which period. For example, it determines which financial transactions are part of which date of business or quarterly summary. It makes that decision within the tally as the facts arrive, regardless of when they occurred in history. The tally is a static model. The central authority uses this static model to guarantee that a transaction is not double-counted, in other words, included in more than one period.

Map-reduce decentralizes the static model. No longer does a single static model have to contain all of the financial transactions that occur within a date of business. Instead, the transactions are distributed among several static models, called shards. To compute an aggregate for a date or a quarter, a coordinator sends a request to each of these shards. The shards each compute their own aggregate, and then shares that result with the coordinator. The coordinator combines all of the aggregates into one final answer. This works because no transaction is ever duplicated between shards. If it were, that duplication would lead to over-counting in the final result.

A block chain is more complex, but avoids the need for a central authority. At many points within the system, individual facts are gathered into candidate blocks. The hash of each block is computed

and tested for some arbitrary condition (e.g. divisibility by a given large number). This arbitrary condition is a proof of work that ensures that satisfactory blocks are found at a desired frequency. Each candidate block contains the hash of its predecessor, and no fact may appear in more than one block in a chain. Nodes within the system will honor the longest chain of satisfactory blocks.

The complexity and inefficiency of block chains make them suitable only in the rarest of situations, where decentralization is of utmost importance, such as for transacting in non-government currencies. Crypto currencies like Bitcoin have pioneered this technique, and continue to be the primary uses of block chains.

When designing a system that requires aggregates over history, add a static model – whether singular or sharded – to the historical one. Model individual transactions historically. At a central authority, collect a list of ongoing historical facts into the static model. At regular intervals, close the tally of facts and compute a summary – either as an aggregate function or via map-reduce. This preserves the logical and technical benefits of historical modeling, while also allowing for aggregation.

As compared to Event Sourcing

Historical Modeling and Event Sourcing are two systems for modeling software behavior. They share a common starting point: that the current state of a system is not the source of truth, but instead derived from the history that lead to that state. Beyond this common core, the two ideas differ greatly in their constraints and subsequent implementations.

Total order vs. partial order

Event Sourcing is the practice of capturing a sequence of events as they affect a given aggregate, and then playing back that sequence to determine the current state of that aggregate at any given time. The sequence of events is ordered. It must be played back in the order in which it was originally captured if it is to produce the desired state.

Historical Modeling, on the other hand, is the practice of representing system state in the relationships among partially ordered facts. The facts know explicitly their predecessors – other facts that specifically came before. But when no predecessor relationship exists between two facts, they could have occurred in either order.

The order of events in Event Sourcing is determined by their container, typically a stream or Event Store. The events don't explicitly know their order relative to one another. But facts in a Historical Model do know about each other explicitly. They determine their own partial order based upon their known predecessor relationships, not based upon their container.

Replay vs. direct query

To determine the current state of an aggregate in an Event Sourced system, start with an object in its initial state. Apply events to that object in sequence to derive the current state. You could think of

an event stream as a mutating sequence of operations that change the initial object into the current one. However, a better way to visualize it is the left fold over the sequence events in a non-mutating manner.

Given an initial state, a0, and the first event e1, you can compute the state after the event is applied using a function f.

```
a1 = f(a0, e1)
```

The function f interprets the event over an initial state, and returns the resulting state. It does not mutate it. We can use this function to compute the state after the next event, e2.

```
a2 = f(a1, e2)
```

Substituting, we get:

```
a2 = f(f(a0, e1), e2)
```

And if we continue the pattern for several events, we get the series:

```
an = f(f(f(f( ... a0, e1), e2) , e3) ... , en)
```

This can be written as:

```
an = foldl f a0 [e1, e2, e3, ... en]
```

A left fold is simply the act of applying a function over a series of events, left-to-right, starting from a given point, and recursively using the result of the previous application. It does not mutate state along the way, but instead produces new state.

The left fold operation must be completed before one can observe the final result. One does not simply examine the sequence of events and answer meaningful questions about the state of the system.

On the other hand, to determine the current state of a Historically Modeled system, you query a fact for its successors. Find all successors meeting a certain condition. That condition is expressed as further queries of the existence or absence of successors. It is not necessary to replay those successive facts to determine the current state. Knowing which successors exist and which do not is sufficient. The existential query is the only operator allowed.

Aggregates vs. facts

Events are replayed against aggregates. The concept of an aggregate is described in Eric Evans' Domain Driven Design. It is a group of tightly related objects (both entities and values). One object, the aggregate root (always an entity), owns all of the others, directly or indirectly. Each aggregate is a tree. A sequence of events led to the state of that tree.

Historical Modeling has no relation to Domain Driven Design. The fundamental building blocks of a Historical Model are facts. There is no separation between facts and entities. Indeed, there are no entities at all. There are only the facts that record that some object was created. To get the state of that object, one does not replay events against an aggregate, one simply has to query the successor facts.

Central state vs. decentralized nodes

Events in an Event Sourced system need to be put in order to be fully understood. Whether the application of events is viewed as a series of mutating operations, or as a left fold over a sequence, order matters. In practice, some events might be commutative. But in general there is no guarantee inherent in the model that any two given events could change positions without affecting the outcome.

The system must therefore identify an agreed-upon order before a set of events can be unambiguously played against an aggregate. To produce this order, and permit no later insertions, requires one of two things: a block chain, or a centralized store. Block chains are extremely inefficient, and justified in only the most specialized of applications, and will therefore not be considered for practical use. Indeed, all of the Event Sourced systems that I've observed use the central store to order the events.

Imagine a system of independent nodes in which each makes decisions and records events. Two of those independent nodes make a different decision related to the same aggregate and record those events locally. Neither knows about the other's event. When they play those events against the aggregate, they get different outcomes, as you might expect.

Now when they share those events with one another, they have to agree on the order. One will be able to append the other's event to its sequence, but the other will be forced to allow an insertion. In some cases, the two events might be commutative, and so the order would not affect the outcome. But in the general case, the commutative property is not guaranteed. And so they couldn't both get away with appending the event.

When playing an event against an aggregate, the state of that aggregate matters. If the aggregate was in a different state, the effect of the event could well be different than intended. When a node allows insertions, it permits the aggregate to be in a different state than it was when the later event was recorded. The effect of the later event might therefore change because of that insertion.

Because each of these independent nodes must be willing to allow insertions, neither can be certain that they have a definitive picture of the state of the system. That one true state can only be known

if insertions are disallowed. In any distributed system, only one node can disallow insertions. And that is why all Event Sourced systems that you find will have an appointed centralized Event Store. Independent nodes will have their own local event stores, but they cannot trust the meaning of any given event until it is in the central Event Store.

Historical Modeling does not depend upon a total ordering of events. It therefore does not require a centralized event store. Facts in a Historical Model can be fully understood at the point that they are added to the local store of an independent node. They don't need to be appended to the stream of a centralized store. The predecessor relationships that are important to understanding the meaning of a fact are part of that fact itself. It is not a property of the container.

The effect of a fact is not to change the state of an aggregate. The effect is merely that a successor exists. Existential queries are the only ones permitted in a Historical Model. This limitation guarantees that all facts commute with others that are not causally related. Two facts, neither of which is a predecessor of the other, will have no bearing on each other's existence.

Consider again the two independent nodes making different decisions with respect to the same aggregate. If they record these decisions as independent facts instead of events, then the story is different. The facts that they record will each refer to the common predecessor. But neither will have the other as a predecessor. How could they, if these two nodes acted without knowledge of the other's decision?

When they query the common predecessor, each will see that their successor fact exists. Eventually they will share their facts with one another. Afterward, the same query will reveal that their successor facts still exist, but now there is an additional successor. They can both see this new state of affairs, but the effect of their individual facts has not changed.

This exchange can take place with no central store, because the facts do not need to be put in order. The nature of a Historical Model guarantees that independent facts are commutative.