Building Large IT Systems

Benedikt Stockebrand

April 19, 2002

Contents

Preface

About this Manuscript

This manuscript is intended to accompany a talk I held on April 19, 2002 in Frankfurt/Main, Germany. It presents one aspect of several I am currently trying to weave into a presentable methodology related to large, reliable and economically efficient IT systems.

I've tried to integrate at least part of the feedback I received during and after that talk into this manuscript. Some ideas however deserve additional research; I have left them for a later time, when I'll be attempting to cover a larger part of the methodology presented. Occasionally it should become obvious that this manuscript is little more than a snapshot of some work in progress and an invitation for comments (best directed to `bs@benedikt-stockebrand.de`). I sincerely hope it will be of some use to you anyway.

Throughout the manuscript I've tried to maintain an image of the atmosphere during that event—including the occasional jokes and snide remarks. If you feel annoyed by them or consider them "unprofessional"[1] you are advised to ignore the footnotes.

Acknowledgments

I thank my teachers at Dortmund University, Prof. Ingo Wegener and Prof. Martin Dietzfelbinger who taught me about all sorts of theory including parallel and distributed algorithms, and Prof. Heiko Krumm, who taught me a lot (if not all) about the not so theoretical problems of distributed systems.

I also thank my colleagues from the "Data Center Management" team at T-Online International AG for uncounted discussions and the good and sometimes bad examples that convinced me that the ideas presented here are not only a practically infeasible theory but an applicable real-world approach towards building large IT systems.

Finally, particular thanks to Wolfang Sachs, Peter Heimann, Martin Pfeilsticker and Gerd Aschemann[2]. They initiated SAGE-Rhein-Main and made me write this paper in preparation for a talk at the first meeting. That very meeting gave me opportunity to gather valuable feedback from a large number of fellow professionals.

About the Author

In the 1990s the author studied computer science at Universität Dortmund, Germany. His major areas of interest were operating systems and computer networks on the practical side and efficient (sequential and parallel) algorithms and computational complexity on the theoretical side. His master thesis dealt with data structures used in symbolic circuit verification, called BDDs, and the computational complexity issues involved.

During his university years he first came into contact with the Internet and learned about the ISP business as a volunteer sysadmin for a local non-profit amateur ISP.

After some system programming and sysadmin jobs he joined the "Data Center Management" team at T-Online International AG, eventually working as system architect and team lead in a large internal project. He will leave T-Online again by

[1] So what's wrong about having fun at work? If we were just doing this for the money we might as well turn pop stars...

[2] In descending alphabetical order by given name. Just for a change.

end of june 2002, after which he will embark on a six month private project entirely unrelated to IT.

Legalese

None of the examples provided here are in any way related to my work at my current employer, T-Online International AG. They are either entirely made up or refer to personal experiences I made in the 1990's and before.

Neither has T-Online International AG been in any way involved with the various activities that led to this manuscript.

1 Basics

Designing and building large systems has become ever more challenging during the last three decades. This text presents an approach to deal with that challenge.

1.1 What are Large IT Systems?

In the context of this paper we consider as *large* IT systems, as opposed to *complex* IT systems as presented e.g. by Glass [Gla98] or Britcher[Bri99], all IT systems that need to span across multiple computers to handle the workload they are built for.

Aside from reliability issues that had to be left out of scope of this text there are basically three reasons that may make a system large:

- There may be no computer available to handle the intended workload.[3]
- Since the bucks-per-bang ratio significantly increases with the size of the hardware used, a single-computer solution may be economically unacceptable. A farm of 256 networked PCs running a suitable Un∗x OS[4] may not match the performance of a high end server (call them Regatta, SF15k, Superdome or whatever), but is easily an order of growth cheaper than the big-iron non-competitors.
- Occasionally the software that the system is based on dictates what hardware can be used, thus limiting the maximum machine size. This restriction usually has one of two reasons: The software may require a specific (low-scale) hardware architecture or it may simply make use of only a limited amount of hardware, like a maximum number of CPUs.

Two common terms related to this are "horizontal" and "vertical" scalability. A system scales "horizontally" if it can be distributed about multiple independent computers and it scales "vertically" if it can make use of multiple CPUs in a single computer. In this context we focus on horizontal scalability.

1.2 The Deferred Split-Up Desaster

Since many IT systems are destined to grow, possibly faster than the performance of the high end hardware, it is only prudent to build a system according to the assumption that it may eventually become "large" according to this definition.

Personally, my first serious sysadmin activity was the re-build of a non-commercial ISP server "farm" where my predecessors had first put everything onto a single machine, then ran into serious performance problems, bought a second machine and tried to move parts of the system onto that second machine. After several months of unsuccessful attempts to move the most performance-critical parts to the new machine it became not only obvious but a generally accepted fact that we were stuck in a dead end: Both machines would only boot together, waiting for each other at certain points. One machine crashing (which happened all too often for reasons beyond the scope of this context) could always count on the solidarity of the other—it would crash, too. And since 100 Mbit/s fast ethernet was still unaffordable at best during that time, the network traffic between those two machines was becoming a bottleneck just as well simply because the work was distributed between the machines without consideration for the network traffic involved..

[3] The incompetence of certain software developers is unfortunately not bounded by the size of the largest hardware available...

[4] Linux is far more popular than the BSD's here. But then, Windows is far more popular on desktop machines than MacOS...

The lesson learned from that experience was that it is virtually impossible to split a one-machine system up so it can be run in parallel on multiple machines. There are inevitably some "minor" details that get missed during the initial one-machine phase that turn up like a terminally sore thumb only when the system is attempted to be split up.

1.3 Inherently Sequential Operations

Theoretical computer science shows that there are classes of problems (their word for "functionalities") that won't parallelize. These problems do have real-world relevance and in some cases you run into them. In many cases however the real-world problem is a special case that can be parallelized with due diligence. But if it happens that your problem is really sequential, the only reasonable approach is to separate them from everything that can be parallelized and have the core problem run on a reasonably big machine. If there is no such machine, you lose.

Similarly, there are problems that can be done in parallel but there is no implementation readily available. Generally, such problems deserve the same approach.

1.4 Large-Scale Functional Separation

The first step when defining the architecture of a large system is the separation of functionally independent components. If there is any chance that a system may become large, there is no reason to put unrelated functionalities on the same machine.

What can possibly go wrong if you as a tiny ISP put a relational data base, an SMTP gateway, a DNS forwarder, and a Tacacs server on the same machine? You may eventually need to split things up.

The first thing that will probably hit you hard is the fact that you need to adjust all sorts of host name configurations. Even if you've been smart enough to provide DNS aliases for all services, like "db(.stupid.example.com)", "smtp", "dns" and "tacacs", all referring to "server", you'll only find out that in a variety of config files you have placed either "server" or the IP address or the wrong alias. This problem sometimes gets "fixed" by using different entries in the `/etc/hosts` on the client machines—the long-term results of this "solution" should be obvious. Things get particularly troublesome if external users are involved. It may be quite difficult to explain to them that no, DNS and SMTP gateway don't need to be on the same machine.

Next you realize that the Tacacs server uses the DB and splitting the two may cause noticeable authentication delays. Unfortunately the tacacs data has long since been "synergized" with the remaining customer data so it is a major effort to pull the tacacs data out of the original DB and build a Tacacs server with its own DB. After all, you now need to keep two separate DBs in sync.

Just after you rebuilt your various Perl and SQL scripts to keep the two DBs in sync you realize that the DNS zone files that need to be updated whenever a customer shows up or leaves won't update anymore: The fancy scripts you've written to create the zone files put the new zone files on the DB server only, leaving the DNS server out of sync. Of course, at least for a "temporary workaround"[5] that's what NFS is made for. Unfortunately, the DB won't fire up without a running DNS server and the DNS server won't work without the NFS server to get its zone files from.

Finally, moving a DNS server to another IP address is one of the last challenges in todays sysadmin world...

[5] You know what that means...

It may be feasible to place several functionalities on the same machine using `chroot` jails and dedicated IP aliases for the functionalities. But once a certain system size is reached, chances are you'll regret you ever tried to. If there is any chance that the system will ever noticeably grow, functional separation is the only way I know to avoid extensive downtime of a by then productive system.

Fortunately, functional separation is a widely accepted approach, not only for large systems but whenever sysadmins successfully keep complexity low to maximize reliability.

1.5 Small-Scale Structural Separation

If functional separation doesn't provide the intended parallelization it is time to split a functional entity. Different to the functional separation the distinction proposed here is in no way canonical or immediately obvious. Instead, it presents a methodology to scale large systems successfully.

Similar to the structural separation of conventional computer architecture into CPU, memory and I/O subsystem, we split IT systems structurally into three kinds of components as shown in figure 1: processing, data storage and communications components. Unlike the functional separation, which was directly related to distribution across multiple machines, the structural separation has another purpose: There are different scalability strategies for these kinds of components. We assume the worst, i.e. that every component needs to scale horizontally over a number of basically constant size subcomponents.

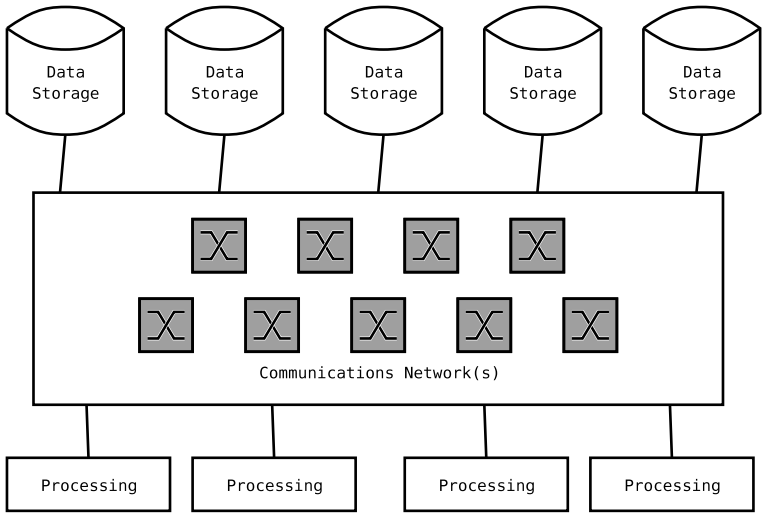

Figure 1: Structural Separation, First Abstraction Level

Again, it may be reasonable to put certain data storage and processing components onto the same machine or set of machines. But the strategies to make the components scale differ and need to be considered separately.

It may be obvious what the communications networks are: These are the various networks between components, like TCP/IP networks and Fiber Channel based SANs. These networks provide the connectivity needed between the other two components. These two are best explained through a generic "work flow": The data components store and maintain some sort of persistent data that is needed for the functionality. The only functionality they need to provide is a consistent,

i.e. transaction-capable or atomic, "read" and "write". The processing components open a transaction with the storage components, do some computations with the data read and possibly some external input, schedule some write operations and execute them by commiting the transaction, possibly generating some externally visible output. If a processing component fails during a transaction, the entire transaction is aborted but can be restarted from scratch, possibly on a different processing component.

If this workflow concept can't be applied, chances are that the system won't scale. In fact, we will see that it is possible that the system may not scale even if the concept applies.

1.6 A Simple Example

Back in the old[6] days when I was an undergraduate student I had to survive a software lab. One of the projects we were doing was a software for a video rental shop. Let's assume we wanted to build another such system, this time for "Video International", a fictional video rental chain operating 24/7 around the world through half a million outlets.

This system would consist of a customer data base, or customer data component according to our definition. For each customer we would save some data, like name and phone number plus a unique customer identifier. The transactions for these components would be creation, modification and deletion of individual customer records (in RDBMS speak) or objects (in OO speak).

Similarly, there would be a data component of video tapes (or DVD's these days), holding the name, current rental fee, and a unique tape identifier. The transactions defined here would be similarly the creation, modification and deletion of individual tape records.

A last data component we'll call the "state" data component, keeps the state of the tape, like "available", or "unavailable until ⟨date⟩", and possibly some reservation made by some customer. The transactions defined here are basically "handing a tape out to a customer until a predefined return time", "receiving a tape back", "reserving a tape for a customer for a given time interval" and "reading out a consistent snapshot from the entire component".

And there would be some processing components. Some components would provide a user interface to create, modify and delete customer or tape data. They would simply separate the data representation and the user access interface. This separation of data and user interface (or "model" and "view" according to the OMV paradigm) is common and doesn't appear too spectacular at this point but proves essential later on: It allows the system to scale the data components horizontally with a minimum of effort.

A similar component would provide a user interface to the state data component so people can reserve, rent out and return tapes. This component is slightly more complex because it doesn't interface a single data component but all three.

To prepare for some oncoming problems, we also want a processing component that attempts some statistic correlation of the "state" data component. This component runs through the entire state data component and tries to do some rather ill-defined statistical analysis.

[6] Well, not *that* old...

2 Processing Components

Picking the easier part of the job first we assume that we already have our data components in place and try to build the processing components.

What is important about processing components and how can they be made to scale? First step during the system design is to define the components. In the "Video International" example we made some decisions apparently at random. Now we start to provide the reasoning behind these decisions.

2.1 Load Distribution and Load Balancing

Assuming that we have a transaction mechanism available with the data components, we first split the processing components into many subcomponents that all run in parallel.

Leaving the transaction/atomicity issue to the data components, the processing subcomponents themselves don't pose any particular problem to the system architecture. Sometimes, as in the case of a GUI processing component, the load on any individual subcomponent is inherently bounded. In other cases it is necessary to provide a load distribution mechanism as shown in figure 2. Several mechanisms are

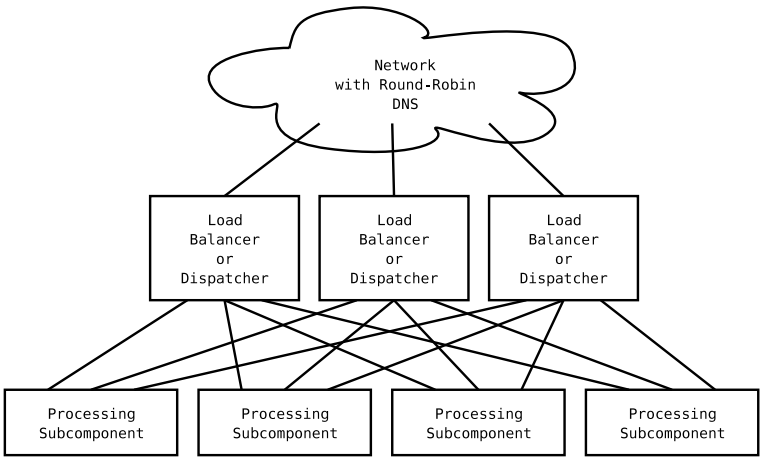

Figure 2: Processing Component Load Balancing

available: Round-robin DNS is a simple way to provide a rough load distribution. Load balancers like Cisco™ LocalDirector™ provide a better distribution and component failure handling. Alternatively, a custom "dispatcher" subcomponent can be written to distribute the load over multiple "backend" processing subcomponents, possibly based on some load measurement on the "backend" subcomponents.

In very large systems, the combination of round-robin DNS and some balancers can be used if a single balancer can't possibly handle the entire load even as a balancer only. While out of the scope of this discussion the approach also proves essential to build reliable systems.

2.2 Pushing Work Towards the Clients

Until now we have ignored the ubiquitous client-server paradigm. Now is the time to assimilate and make good use of it.

A client is little more than a processing subcomponent that may access the data components either directly, like an Oracle™ client connecting through SQLnet™, or indirectly through other processing components, like a web browser connecting through a web server and CGI script to some data files. So the concept presented doesn't interfere with the client-server paradigm.

In a client-server environment the load on the server normally grows monotonously, in most cases somewhat linearly, with the number of clients. Since server hardware is usually more expensive than client hardware and client hardware by definition scales horizontally, it is good practice to push as much load as possible towards the clients. Put flatly, Java™ applets are a better idea than Java™ servlets—at least until the network components turn out to become the most pressing problem or the clients need excessive sysadmin care.

At least in a LAN environment it may be perfectly reasonable to push all processing components towards the clients, eliminating the need for load distribution and load balancing mechanisms as proposed previously. Both the network and the sysadmin effort problems can often be overcome by caching the application on the client. This eliminates all unnecessary network traffic, since the application is only transmitted again if the cached version is out of date. And caching provides a software "deployment" mechanism that requires virtually no sysadmin care at all on the clients.

2.3 Crash Recovery

Processing components not only scale well, they are also easy to make redundant.

If a processing subcomponent fails while working on a job, the entire transaction must be considered lost and the job needs to be restarted. This can easily be implemented provided that the transaction concept is sound.

Similarly, the data component that a transaction is opened with needs to check the livelihood of the processing subcomponent whenever a competing transaction tries to access a locked piece of data. The processing component only needs to provide an interface for this livelihood check.

2.4 Transaction Size Problems

Things can get difficult if the transaction size grows with the load, or the amount of data stored in the data components. We will take a closer look at the transaction problem when we analyze the data components and now only focus on the processing component side of the problem.

As far as the processing subcomponents are concerned it is important that any job can be run on fixed size hardware. In the case of "Video International", the transactions for user, tape and reservation manipulations are effectively of constant size and don't pose any particular problem.

The ill-defined statistic correlation component however accesses the entire state data component. A simple read-process-write activity will need network bandwidth and memory increasing with the amount of data stored in the state component. This makes it necessary to break down the component into subcomponents using a multi-level approach. A "master" subcomponent level provides the "statistics" interface and controls a set of "slave" subcomponents that run in parallel and work on parts of a single job coming in from the "master" subcomponent. If it is impossible to break up a functionality this way the functionality itself either doesn't scale or needs some special analysis beyond the scope of this text.

2.5 Layered Processing Components

Using a layered approach won't be useful when dealing with large transaction size only but as a fairly generic approach.

The client-server model we've seen in section 2.2 has shown another good use for layered processing components.

Similarly, computation-heavy functionality can be attempted to be split in a layered manner. This is particularly useful if different layers have different hardware requirements that allow for somewhat specialized hardware. These can be simply CPU vs. memory heavy layers or layers that use e.g. special crypto hardware like the SSL accelerators available from some vendors.

Systems built out of standard software components also tend to break up easily into layers of components. A web server running CGI scripts can be considered consisting of a layer of web servers like Apache and another layer of CGI scripts.

Finally, in a reasonably complex system it may be clever to provide these layers of processing components to break up the system into independent parts. While many software developers still don't really appreciate—and use—the possibilities a distributed system provides them with, some do. After all, parts of a job may be more easily implemented in one programming language while other parts are asking for another. Shell scripts, AWK and Perl are easily modifiable for not too performance-critical jobs, Java has some fancy features that can be useful occasionally, C provides a clean interface to the underlying operating system and C++ is the choice language if a big system needs to be implemented, providing for both an object oriented approach and performance[7].

3 Data Components

Making processing components scale was fairly straightforward. Data components however prove more of a challenge. Transactions, the blessing of todays RDBMS, turn out to be the bane of scalability.

3.1 Splitters and Leaf Subcomponents

The most common way to structure a data component is to provide a sufficient number of "leaf" subcomponents that hold the data and a "splitter" (sometimes called "director") subcomponent or several "splitter" subcomponents that know what data is stored at which leaf through an indexing mechanism. All access from outside the component is done through the splitters.

This is pretty much like a library with lots of bookshelves (the leaf subcomponents) holding books (the data) sorted by author (the index), and at least one librarian (the splitter) who knows how the index relates to the bookshelves.

Implementation of the index can be done through actual index subcomponents. Alternatively a hash function may be used. While the former approach allows fairly easy redistribution of data in a running system and localization of data in a geographically distributed environment, the other performs and scales better. The choice is largely a matter of taste and the particular requirements. Figure 3 shows an example data component setup that makes use of an index subcomponent.

3.2 Transactions

What is the problem with transactions? Why don't they scale?

[7] . . . at least if you know what you are doing. . .

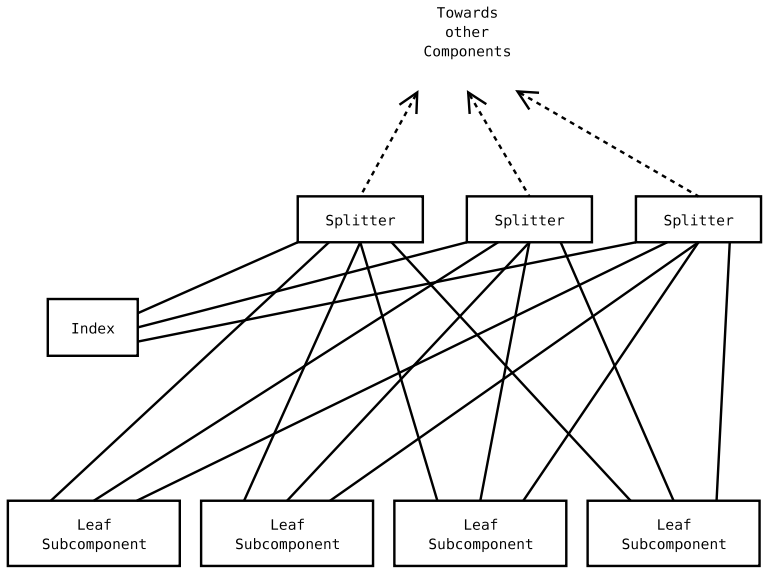

Figure 3: Generic Data Component Structure

The easy answer is that the transaction mechanism in general is too powerful. It can't be parallelized properly, at least in a real-world environment where individual components sometimes fail.

Imagine a set of transactions that spans multiple data components, each implemented through multiple subcomponents in itself. Where can we maintain that transaction information? We need to put the transaction where the data is so they don't get "separated" from each other, leading to inconsistencies that we want to avoid at all costs. But then, that transaction needs to spread across multiple components or at least subcomponents in itself because it can't be limited to a single subcomponent in general. This directly collides with the all-or-nothing nature of transactions: If one of the components involved becomes temporarily unavailable all other components involved get stuck since they can't always reliably know about the state of the failed component until it comes up again. The theory behind this issue based on an abstract problem called the "Three Army Problem" (or sometimes "two-army problem") and explained by Tanenbaum [Tan96, p.499f].

Even without these reliability-related problems the transaction mechanism emulates a sequential manipulation of the data stored. This emulation requires some sort of central control that defines the sequence in which the manipulations are arranged. While this observation has fairly little real-world relevance in itself it shows that there is a fundamental problem with transactions.

So how can we deal with this problem? There are two options, none of which may be applicable to a given real-world problem, though: We may either try to get away with transactions in general or we may restrict transactions in such a way that they don't need to be distributed across multiple machines.

3.3 Static and Semi-Static Data

The easiest way to avoid transactions is to eliminate all write operations from the processing jobs. This of course can't be done in general, but if we identify all static data as such we can scale a possibly large part of a system.

Static data can be easily replicated and scales well. Just put as many copies

on as many identical subcomponents as necessary. If you run out of capacity, add more subcomponents.

Now data base programmers may complain that in order to use SQL to its fullest they need the static data in the same RDBMS as the dynamic data they are working on. This is fine unless the data base turns out to be critical to scaling, in which case everything that doesn't have to be in that data base needs to be ripped out—the static data as well as complex SQL statements and stored procedures.

In our "Video International" example consider a table mapping zip codes to town names. We will see soon that the customer data component can be fairly easily scaled across multiple machines. There is absolutely nothing wrong with keeping copies of that zip code table with every customer data subcomponent, except possibly some performance aspect. Keeping the zip code table a separate component from the customer data component doesn't mean they can't be located on the same hardware and in the same table space. We just need different approaches to make these components scale. Static data like the zip code table is only once copied where needed while the customer data needs to be taken particular care of.

Every once in a while, zip codes actually change. If that happens, our decision to consider zip codes static data causes two problems: It may be necessary to bring the entire system down for an update and the change may cause some inconsistencies throughout the system if the developers assumed that zip codes are perfectly static.

These limitations may be considered acceptable or not, depending on the functional requirements that the system has to meet. "Video International" may accept a five minute shutdown every couple months to update its zip code table, especially if scheduled downtimes at night are acceptable. An official Internet-based zip code data base may want to avoid even that downtime.

Keeping the downtime to a minimum can often be achieved by first copying the new data to the operational subcomponents but not using it yet, then shutting the system down shortly, substituting the data on the subcomponents and then bringing the system up again.

3.4 Monotonous Data and Caching

But even if data is incessantly written it is sometimes possible to avoid transactions. The most prominent case is "monotonous" data, i.e. data that only once gets created and never actually changes.

Consider our tape data component from "Video International". If we are bright enough to stick to time-proven techniques we assign each tape a unique tape ID. And that ID won't be recycled for whatever other tape, even if the first tape gets thrown out—after all, IDs aren't that expensive.

Doing so assures that certain data won't ever *change* but will only get added. This effectively makes it unnecessary to protect against changes through transactions. Data can be replicated through caching. Whenever a cache subcomponent is asked for some data it first searches its internal cache and only if that proves unsuccessful it asks a "central", "authoritative" data subcomponent. Cached data can be preserved for an unlimited time and normally gets thrown away only when the individual cache subcomponent runs short of storage.

Of course this eventually clutters up the authoritative data subcomponent. Old data that has become irrelevant, like tapes that have been removed from stock and long since thrown away should be removed from the data subcomponents. While the caches can be left to their own device it may be necessary to sanitize the authorative data subcomponent. Data that is obsolete however shouldn't be involved in a transaction anymore, so some sort of "sanitization" or "garbage collection"

mechanism can be implemented in a fairly straightforward manner and without the need for transactions.

At this point it should have become obvious why we separated the tape data component from the state data component: We were hacking off the parts that we can provide an easy solution for, trying to make the hard core of the problem manageable.

3.5 Directories and Temporary Inconsistencies

Sometimes it is possible to avoid transactions by explicitly dealing with temporary inconsistencies. This line of approach is common with "directories", data bases that are optimized for read access and don't provide transactions, like LDAP or DNS.

With our "Video International" example the customer data component falls into this category. Sometimes customers change their name or address or even stop renting videos. These changes are infrequent but do happen. A changed name or address is not particularly critical; even if a bill may be sent to the old address several minutes after the address was changed this may not warrant the use of transactions.

To avoid serious inconsistencies at this point it is however essential to read all data only once during an operation and to assure that all the data read is in itself consistent. This will avoid using both the old and new address for a customer, for example; sending a bill to the customers old address is fine, but using the street name from the old and the zip code and town from the new address is bound to cause trouble. This may be difficult if large parts of data components need to be read consistently. In this case a "lightweight" version of the transaction mechanism as proposed in section 3.8 can possibly be applied. This "lightweight" transaction mechanism only needs to provide for "read" locks but doesn't need "write" or "commit" functionality.

There are several well-known ways to make a directory scale. Looking at the way DNS is implemented shows that a well proven approach is to provide a "primary" subcomponent that all write operations are directed to, some "secondary" subcomponents that serve all the read requests, and a mechanism to synchronize the secondaries against the primary. To provide high reliability however, more complex alternatives out of the scope of this text may need to be used to avoid having a "primary" subcomponent turning a single point of failure. Additionally, caching may be a possibility if a time-to-live can be defined for all data such that data outdated less than that time is acceptable to be used.

3.6 Failing Towards a Defined Direction

There are cases where transactions are unnecessary because a failure in a single direction is acceptable.

A well-known example of this approach is the way the SMTP protocol is designed. A message may never be lost during communications but may possibly be sent multiple times. This example doesn't cope with the loss of data while stored in an SMTP host. In general however, a redundant setup that will only fail in a single direction due to a hardware failure is equally possible.

If we are dealing with data items that get stored somewhere, like e-mail messages or Usenet news articles or such, it is possible to use unique (message) IDs to sort out multiple transmissions at the leaf data components[8].

[8] In the case of SMTP such a "deduplication" feature would of course be a fatal security flaw.

While the SMTP example is limited to communications reliability it shows how to avoid transactions in some cases, converting an "exactly once" transaction into an "at least once" transaction.

3.7 Transaction Space Partitioning

If transactions are actually necessary the next step is to see if the transactions only involve disjoint data sets. If the transaction space can be partitioned into reasonably-sized and disjoint parts it is possible to provide transaction-capable data components that actually scale.

In our "Video International" example we have managed to get away without transactions as far as the customer and tape data components are involved. Tape reservations however do need transactions: We cannot afford to have multiple customers reserve the same tape for the same time slot. But then, tape reservations only relate to a single tape, so we can partition the transaction space for the tape state data component by tape. The transactions can thus be pushed towards the individual state data subcomponents.

This approach is extremely simple, reliable and efficient if it is applicable. The hierarchical transaction spaces presented in the next section shouldn't even be considered until a "flat" transaction space partitioning has proven to be unapplicable.

3.8 Hierarchical Transaction Spaces

What can we do if the transaction space can't be partitioned into manageable chunks? Precious little actually, unless we are willing to spend quite some effort on it.

If we can't limit transactions to individual leaf subcomponents we need to push the transactions up towards or even above the splitter subcomponents involved. We need to introduce special subcomponents that only maintain transactions and pass all operations through to the splitter or leaf components below. In RDBMS speak these subcomponents are called "transaction monitors"[9].

Applying his approach without the use of existing implementations is complex and error-prone, especially as soon as reliability issues get involved, because error-handling gets more complex since it has to deal with partial failures. If scalability is not too much of an issue it may be feasible to use only a single splitter that keeps all transactions. This will improve scalability by separating the splitter and transaction part of the data component from the actual data storage leaf nodes. If the splitter can't handle everything it may be possible to use multiple layers of splitters, especially if transactions are comparatively large.

This approach requires a huge effort to achieve a small increase in scalability. It should only be considered a last resort. This explains why the statistics feature in our "Video International" example is so troublesome—unless we accept some inaccuracies because we won't insist on a perfectly consistent snapshot of the state data.

3.9 Distributed Transactions

Why do transactions need to be located at a single subcomponent? Shouldn't it be possible to have a distributed transaction mechanism?

The short answer to that question is "no", according to the theory behind the "three army problem" mentioned above. It may be reasonable to come up with

[9] Thanks, Gerd.

a weaker transaction definition that can be used in many cases but does scale horizontally, but to my knowledge no such concept has yet made it towards a generic and industrial strength[10] solution.

The problem is basically that first you develop a solution running in a 100% reliable environment. Then you need to implement some error recovery since the environment isn't 100% reliable. Then you need some error recovery recovery in case something goes wrong during error recovery and so on to infinity and beyond[11]. Even the immediately obvious approach "deal with errors beyond a given error layer manually" is tricky, since the error recovery layers often don't relate to the "severity" of an error; some simple real-world error may make its way straight down an arbitrary number of error recovery layers if the error model that the error recovery is based upon doesn't closely match the real-world reliability of the components involved.

But if we leave reliability issues aside it is actually possible to recycle a locking mechanism concept that is used in another context. What we need are multiple transaction subcomponents that need to arrange for a distributed locking mechanism.

This problem is effectively the cache handling issue in SMP[12] hardware in a different guise. As long as all CPUs access different memory areas, things perform reasonably well. Once they try to access the same memory areas however, performance significantly degrades. What's worse, cache management doesn't scale well over too many CPUs.

We can try to implement the SMP cache management to provide a distributed locking mechanism. A reasonable line of approach may be to provide multiple levels of transaction locks. Consider the state data component of "Video International" and the statistical analysis processing component. Assume that the statistics component needs to obtain a consistent view of the data. This implies that a processing component needs to obtain a lock, or in this case at least a consistent read, on all data within the state data component. This could possibly be handled through such a distributed locking mechanism.

Again, this approach provides only limited scalability and no serious transaction behaviour in real-world systems where subcomponents occasionally fail. A line of approach like "if anything goes wrong stop everything and call the sysadmin for help" is rarely a solution., at least once the number of machine exceeds the "pain threshold".

3.10 Detour: EAI Middlewares

During the last years, "Enterprise Application Integration" (EAI) and "Middleware" became popular buzzwords. What are these and what about the occasional claims that they provide a "reliable" communications mechanism? We now have the tools available to verify some of the statements often heard from EAI vendors.

EAI software generally claims to provide two features: A reduced number of intercommunication links between applications and an additional abstraction layer between the applications and the network protocol stack layers, providing "reliable" communications. Figure 4 shows a sample EAI environment with four machines and three applications (which are hopefully implementing either data or processing components).

[10] "Industrial strength" here means "applicable in a real-world environment", not "commercially sold by a vendor that can be blamed if/that it doesn't work".

[11] In case you wonder: This is Buzz Lightyears (of "Toy Story") motto.

[12] SMP = Symmetric MultiProcessing, i.e. computers with multiple CPUs

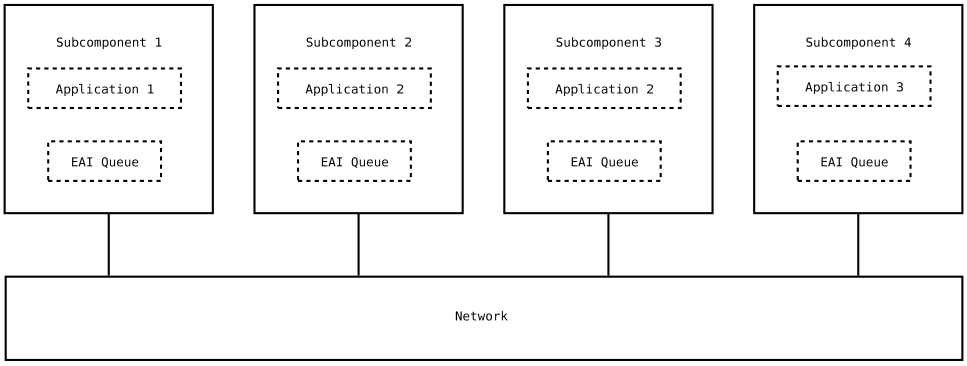

Figure 4: Sample EAI Environment with three Applications

The first feature is based on the idea that all applications only communicate with the EAI software which takes care about where to send which data. This is fine from a syntactical point of view, and aside from possible performance effects it is definitely a good idea to use a standardized data format for communications between applications, be it ASN.1/XDR, Java RMI or XML, and a centralized "routing" management. Making different applications work together still requires a lot of "adapter" programming though; in this respect EAI is little more than a methodology rather than a shrink-wrapped solution.

The second feature is more interesting: EAI allows to queue messages to other applications "reliably" and assures they are eventually delivered. There are two important implications of this approach: A "serious" failure may lose the queue, possibly leading to inconsistencies since some messages related to a "transaction" may have been lost while others have been executed. That failure may be a hardware failure, a buggy operating system or system software or simply a bug in the EAI software. In any case, there may be undetected half-finished "transactions" left from such a failure, leading to data inconsistencies between applications.

Similarly, there is no such thing as a "transaction", since a middleware can't guarantee that a set of messages presenting a "transaction" has that "all-or-nothing" property a real transaction has. While messages are sent, some arrive earlier than others, leading to a transient inconsistency. As soon as network reliability must be taken into consideration, these "transient" inconsistencies can last as long as a network failure. What's worse, they can't be reliably detected since our considerations about error recovery from last chapter still hold.

EAI software simplifies several important and tedious tasks involved in distributed software development, much like relational data bases do in their domain. But it doesn't solve the transaction problems we've seen and as such doesn't even address most of the problems we've encountered in this chapter.[13]

4 Communications Components

Finally, we need to take care of the communications components involved. Aside from the reliability considerations out of scope of this text it is the communications among the software subcomponents that dictate on which machines to locate the processing and data subcomponents. So after we've dissected the system and split it into data and processing components, and then split those in subcomponents,

[13]In case you've been in Frankfurt during that talk and can't recall me talking about this: I've skipped this topic from the actual talk due to time constraints.

the communications components make us put those structural subcomponents back together on real machines.

4.1 Estimating Upper Bounds

The first step is to eliminate upper bounds on communications. This helps to put communications bandwidths and latencies into proportion.

At that non-commercial ISP mentioned in section 1.2 someone proposed to replace the Thinnet we were using there with a switched Fast Ethernet network. Aside from the fact that exactly one machine could have been equipped with Fast Ethernet this bandwidth would've been entirely oversized compared to the obvious upper bound defined by the sum of the single 64 kbit/s uplink and eight analog modems at 14.4 or 28.8 kbit/s each. It took some time and effort to explain that the bandwidth provided wouldn't be needed and it took even more time and effort to explain that no, the latency wouldn't noticeably change since the latency between the users modem and the ISPs modem was about 180 ms, the latency within the Thinnet was around 4 ms and the latency upstream along the 64 kbit/s line was another 120 ms...

Similarly, the traffic between (sub-)components can often be estimated. In particular, the data traffic passing between the splitter subcomponents and the processing subcomponents is usually roughly equal to the data traffic between the splitter subcomponents and the leaf data subcomponents. Such relative bounds, explaining the relation between the bandwidths used between components and subcomponents, are usually at least as important as absolute bounds; they explain where the system will be heading when it starts to grow.

4.2 Bandwidth Tuning

This last observation helps to locate the subcomponents: Since we can't keep the splitters close to the leaf data subcomponents it is usually smart to put splitters towards the processing subcomponents to keep half the communications traffic local within the machines involved—provided that it is possible to have multiple splitter subcomponents. And since the division into subcomponents is largely a conceptual one, it is perfectly reasonable to have splitter subcomponents and processing subcomponents even within a single binary as long as they are conceptually separated and there can be a one-to-one relation between them.

Things can get significantly more complex if WAN connections get involved. In this case, as a general rule as much data as possible should be replicated or cached in each location. A more detailed analysis of the particulars of a given system at this point virtually always proves worth the while, since especially in WAN networks bandwidth is too expensive to be wasted.

4.3 Latency Tuning

Another problem that often becomes important within a geographically distributed environment is latency.

Generally, software developers should take into consideration latency issues by minimizing the number of communication rounds. But if we consider the software as "fixed" it is important to take care of latency as well as bandwidth.

Bandwidth and latency behave fundamentally different: Bandwidth increases with the load of a system while latency stays mostly constant. And bandwidth can be increased if someone is willing to pay for it; latency is largely determined by the distances involved and the speed of light. Technically this property puts latency

out of the scope of a "large systems" discussion but just like bandwidth it is vital to know and understand.

The tuning measures are similar to bandwidth: Put those components that are latency critical together as close as possible—and keep prodding the software developers to use as few communication rounds as possible, especially on high-latency connections.

5 Related Topics

Throughout this text we've mostly avoided to discuss reliability issues. But reliability is critical to large systems since the larger a system the higher the probability that components fail.

Reliability in itself is a hugely complex subject, far more complex than scalability as presented here. What's worse, reliability has many distinct aspects that often interfere with each other, while scalability is usually either a reasonably straightforward problem or an unsolvable one.

Similarly we have ignored performance except when we finally talked about communications components.

Another topic we've entirely ignored is the question how to operate a large system. Again, this is a complex subject in itself. Fortunately, the approach presented at least doesn't inherently interfere with operatability. Providing many small but functionally equivalent subcomponents allows for efficient operation for two reasons: First, machines can be categorized by "classes", helping to organize work in such a way that much work can be done once at the "class" level instead of any individual machine through tools like `cfengine`, `rdist` and the occasional shell script one-liner[14]. Second, if reliability issues have been properly taken care of it is possible to bring down most machines for maintenance purposes without disrupting service.

The shortlived and technology dependent choice of hardware and (system) software has been entirely ignored as well. This is only partially because of the shortlived nature of any answer to this question. More importantly the choice of "platform" or "infrastructure" components should primarily be based on the local "strategic platform", not on the particulars of any single system. Even if the strategic platform is insufficient to cater for the needs of a single system that platform should be reconsidered according to strategic considerations, taking the particulars of the system only as hints as where to extend or update the strategy.

Finally, the political hassles involved with large systems and their design have been blissfully ignored. Among others Adams [Ada96] [Ada97], DeMarco [DeM95] [DeM01], DeMarco and Lister [DL99], Jones [Jon94], Machiavelli [Mac14][15], Roberts and Roberts [RR00], Weinberg [Wei86] and Yourdon [You97] have some useful insights I've personally found helpful myself[16].

References

[Ada96] Scott Adams. *The Dilbert Principle*. Harper Collins, 1996. One of the more enjoyable books on hi-tech projects and geek life in general. But the insights Adams presents aren't just fun—the OA5 concept alone is worth some serious thinking.

[14] Like "`for h in $HOSTS; do ssh $h init 5; done`" where "`HOSTS`" has been previously defined as "`[a-m].root-servers.net`".

[15] This is dead serious. If you're running out of realism this will provide you with more than you can probably bear.

[16] Gerd, maybe you'll find an answer to your problems in one of these.

[Ada97] Scott Adams. *Dogbert's Top Secret Management Handbook*. Harper Collins, 1997. Similar to [Ada96].

[Bec00] Kent Beck. *Extreme Programming Explained—Embrace Change*. Addison-Wesley, 2000. One of the very few books on extreme programming so far. The methodology of XP is mostly a collection of ideas generally not accepted by most management, interwoven into a consistent methodology. The three pages on risk management alone may be worth the book, especially if management can be made to actually read them.

[Bri99] Robert N. Britcher. *The Limits of Software*. Addison-Wesley, 1999.

[DeM95] Tom DeMarco. *Why Does Software Cost So Much?* Dorset House, 1995. A loose collection of independent articles covering a wide range of issues from management to peopleware to technology and methodology issues.

[DeM01] Tom DeMarco. *Slack—Getting Past Burnout, Busywork and the Myth of Total Efficiency*. Broadway Books, 2001. You can't drive at top speed and to a u-turn at the same time, no matter what.

[DL99] Tom DeMarco and Timothy Lister. *Peopleware—Productive Projects and Teams*. Dorset House, second edition, 1999. Standard reading. Also somewhat useful as a projectile to aim at a manager who tries to "manage" a brainworker team like the oarsmen in a man-o-war. (Don't actually throw it though, the book's too good to deserve such gruesome fate.).

[Gla98] Robert L. Glass. *Software Runaways*. Prentice Hall, 1998.

[Jon94] Capers Jones. *Assessment and Control of Software Risks*. Prentice Hall/Yourdon Press Computing Series, 1994. This book presents "traditional" risk management, i.e. addressing risks by watching and taking countermeasures when they become troublesome. While I consider this approach a second line of defense only after the risk minimization approach described by Beck [Bec00] this book most definitely holds a treasure of quantitative facts.

[Mac14] Niccolo Machiavelli. *Il Principe (The Prince)*. 1514. Todays big companies are not really that much different from Italian renaissance city-states. Personally I particularly like chapters 12 and 13, concerning the use of mercenaries and borrowed (aka. "auxiliary") troops.

[RR00] Sharon Marsh Roberts and Ken Roberts. Do I want to take this crunch project? In *[WBK00]*, pages 25–42. Dorset House, 2000. If you need to survive a project with a desperate time line this is quite useful. If you need to figure out if management is really as desperate as you are told it is even more useful. If you consider this article interesting you may also want to take a look at [You97].

[Tan96] Andrew S. Tanenbaum. *Computer Networks*. Prentice Hall, third edition, 1996. A standard textbook on computer networks.

[WBK00] Gerald M. Weinberg, James Bach, and Naomi Karten, editors. *Amplifying Your Effectiveness*. Dorset House, 2000. Some excellent essays about technical work and project management.

[Wei86] Gerald M. Weinberg. *Becoming a Technical Leader—An Organic Problem-Solving Approach*. Dorset House, 1986. Excellent reading when you suddenly find yourself being called a "team leader" and are expected

to organize a group of people into a technical team. Especially useful when that group consists of some consultants new to the company and is brought into a huge project when it's already half way through....

[You97] Edward Yourdon. *Death March—The Complete Software Developer's Guide to Surviving "Mission Impossible" Projects*. Prentice Hall, 1997. A disillusioning analysis of the way software projects are managed today, covering incompetent management, politics, irrationality, survival as a project member and some suggestions how to improve the situation. Most recommended reading if you're just about to quit your job from frustration—not because it'll stop you from doing so but because it'll help you understand your situation more clearly.