32. System Administration

The goal of system administration (admin for short) is to reduce the customer's total cost of owning a computer and keeping it working (TCO). Most of this cost is people's time rather than purchased hardware or software. The way to reduce TCO without giving up functionality is to make software that does more things automatically. What we want is plug-and-play, both hardware and software, both system and applications. We certainly don't have that now.

There are two parts to admin:

- Establishing policy: what budget, what priorities, how much availability, etc.
- Executing the policy: installation, repair, configuration, tuning, monitoring, etc.

The problem

The customer's problem is that too much time (and hence too much money) goes into futzing with computers: making them work rather than using them to do work. Companies and end-users have different viewpoints. Companies care about the cost of getting a certain amount of useful computing. End-users care about the amount of hassle. They can't measure the cost or the hassle very well, so perception is reality for the most part.

Companies

Corporate customers want to reduce total cost of ownership, defined as all the money spent on the computer that doesn't contribute directly to productive work. This includes hardware and software purchases, budgeted support people, and most important, time spent by end-users in 'futzing': installing, diagnosing, repairing, and learning to use the computer, as well as answering other users' questions.[1]

Most of TCO is support cost and end-user futzing with the computer, not purchase of hardware and software. At least two studies have tabulated costs of $8k/year, distributed as follows in one of them:[2]

Hardware	$2,000	24%
Software	$940	12%
Training	$1,400	17%
Management	**$3,830**	**47%**
Total	$8,170	

[1] Perhaps it should also include time the user spends playing games, surfing the Web for fun, and writing love letters, but that's outside the scope of this handout.
[2] Forrester, quoted in Datamation, June 1 1996, p 10. There's a similar study by Gartner Group that reports about the same total cost/year. I've also seen $11k/year!

The management cost breaks down like this

End-user downtime	$1,350	35%
Desktop administrator/tools	$1,280	34%
Coworker time	$540	14%
Disaster prevention/recovery	$660	17%

It's hard to take the three significant figures seriously, but the overall story is believable. Certainly it's consistent with my own experience and with some back-of-the-envelope calculations.

Another example of the dominating cost of admin is storage. It costs $100-$1000/year to keep one gigabyte of storage on a properly run server, with adequate backups, expansion and replacement of hardware, etc. To buy one gigabyte for your PC costs about $30. So the cost of purchase is negligible compared to the cost of admin.

End-users

End-users don't think in terms of dollars; they just want the system to work without any hassles—not such an unreasonable demand. Put another way, they want less futzing and more time for real work or play. Of course, they also want a fast system with lots of features.

Here are the problems users have that lead to futzing:

Install	I plugged it in and it didn't work.
Repair	It worked yesterday, but today I can't ...
Replicate	I can't get to a copy of ...
Reconcile	I have two copies, and I just want one.
Convert	I have a Word file, but my Word won't read it.
Learn	I don't know how to.... I did ... before, but how?
Find	I can't lay my hands on ...

The ones above the line can be addressed by better system administration. The ones below need better usability, help, and information management tools.

Architecture

Admin is what is left over after the algorithms programmed into all the system components have done their best. This description implies a modular structure: components doing their best, and admin controlling them.

Ideally, the components would adapt automatically to changes in their environment. Here are some examples of components that work that way. Most of them are network components.

Ethernet adapters, hubs, and bridges (switches).[3]

IP routers within a domain, using OSPF to set up their routing tables.[4]

The AN1 network (see handout 22).[5]

The Petal disk server.[6]

A replicated storage server.[7]

Today admin is almost entirely manual, both for setting policy and for executing it. Setting policy has to be manual, since it's the way the system's owners express their intentions. We want to make execution automatic, so the only manual action is stating the policy, and software does everything else.

Policy has two parts:

- Registering users and components
- Allocating resources, by establishing quotas, setting priorities, or whaterver.

We won't say anything more about this.

What does execution do? It keeps the system in a good state. To make this idea useful we need to:

- Define the system state that is relevant for admin. This is a drastic abstraction of all the bytes on the disk and in memory.
- Describe the set of good states. This is partly defined by the user's policy, but mostly by the needs of the various components.
- Build software that gets the system into a good state and keeps it there.

Describing good states is better than giving procedures for changing the state, because you can easily check that the system is in a good state and report the ways that it isn't. Often you can also compute a cheap way to get from the current state to a good one, instead of redoing a whole installation. More generally you can analyze the description in many useful ways.

This section describes how to organize a system this way, how to define the state, and how to describe good states using predicates.

[3] R. Perlman, *Interconnections: Bridges and Routers* , Addison-Wesley, 1992, chapter 3.
[4] Perlman, chapters 8-10.
[5] Autonet: A high-speed, self-configuring local area network using point-to-point links, *IEEE Journal on Selected Areas in Communications* **9**, 8, (Oct. 1991), pp1318-1335.
[6] E. Lee and C. Thekkath, Petal: Distributed virtual disks, *Proc. 7th ACM Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1996, pp 84-92.
[7] B. Liskov and B. Oki, Viewstamped replication: A new primary copy method to support highly available distributed systems, *Proc. 7th ACM Conference on Principles of Distributed Computing*, Aug. 1988.

Organizing the system: Methodology

How should we organize a system in order to minimize its total cost of ownership without hurting its functionality and performance? I've divided this question into three main topics: modularity, large scale admin, and certification.

We have two kinds of modules, *components* that do real work, and *admin apps* that coax or bully the components into states that meet each other's needs and carry out the user's policy. Components let admin read and change the state and track performance. Admin builds a model of state and performance, tells components what to do, and tells the user about the model.

For the admin app itself to be used effectively on a large scale, it has to be able to handle and to summarize lots of similar components automatically. Large scale admin requires some way to move lots of bits; this requires connectivity, but many combinations of CD-ROM and network connections can work.

It would be nice if the architecture were perfect and everyone implemented it perfectly. Since that won't happen, we need a way to certify components and sets of components that are easy to administer, so that customers can make rational choices.

Modularity

Today admin is almost entirely manual. It includes:

Allocating resources (disk space, IRQ's, etc.).

Organizing the file system name space and (usually) deciding where to put each file.

Establishing and implementing a security policy.

Replicating things (for backup, onto multiple machines, into multiple documents, etc.).

Installing and upgrading hardware, software, fonts, templates, etc..

Configuring components to fit the environment (network stuff, I/O devices, etc.).

Diagnosing and repairing broken hardware and software.

Finding things.

The tools that allow you to do these things also give you lots of opportunities to get it wrong. We want admin to be automatic. That means it needs to be modular, separate from the components; for this to work, the components have to provide some help.

If components took care of everything, we wouldn't need any separate admin (well, a little for setting policy). Certainly it's good for a component to be more self-managing, but it's not practical to solve the whole admin problem this way. We need the modularity of separate components and admin for several reasons:

- Developing components is hard enough; it mustn't be held back by admin development. What we want is the opposite: to be able to work on admin separately from working on components.

- Components have to be coordinated, and it's easier to do that from one place than to have them all talk to each other.
- Different kinds of admin are appropriate for different situations. You don't want to manage an aircraft carrier and a dentist's office in the same way.
- Automatic admin is fairly hard, which means it has to be done with common code, either services that components call or admin apps that call the components. Services are a lot harder to design.
- It's easier to change an admin app than to change a lot of components or to change services that many components call.
- No single vendor controls all of the components, and it takes a long time to get the whole industry to change its practices. This means that you have to work with legacy components.

How are responsibilities divided between admin apps and components?

An admin app builds a model of the state, updates it efficiently to reflect changes, understands the predicates that describe the good states, and tells the components to make state changes so that the predicates hold. It takes care of completing or undoing these changes after failures if necessary. It presents the state to the user and accepts instructions about policy. Finally, it observes and models performance and presents the results to the user.

Here are three examples of admin apps that together suggest the range of things we want to do:

A diagnostician that tells you what's wrong or strange in your system.

A software installer that keeps a set of apps correctly installed and up to date.

A hierarchical storage manager that automatically backs up and archives your files.

A component allows every bit of state to be named and provides operations to read the state (and track it efficiently, using change logs; more on this later) and to update the state. The Internet's Simple Network Management Protocol (SNMP) is an example of how to do this; it was designed to manage networking components, but can be applied more widely. Admin has to be able to make a set of changes reliably, even a large set that spans several components. This means that the update operations must have enough idempotence or testability that admin can implement redo or undo. The update operations must also be atomic enough to meet any requirements for the system to provide normal service concurrently. If these are strong, the components will have to do locking much like transaction processing resource managers. Usually, however, it's OK to do admin more or less off line.[8]

Modularity means interfaces, in this case the interfaces between components and admin. Like all interfaces, they need to be reasonably stable. They also need to be as uniform as possible; one of the reasons for the states-and-predicates architecture is to provide a standard framework for

[8] If all the components implement general multi-component distributed transactions, that would do the job very nicely. Since it's unlikely that they will, it's a good thing that it's unnecessary. Admin can take on this responsibility with just a little help.

designing these interfaces. Currently, the state of the art in interfaces is rather primitive, well represented by the Internet's SNMP, which is just a version of the naming interface defined in handout 12.

Of course everything in computing is recursive, so one man's admin app is another man's component. Put another way, you can use this architecture within a single component, to put it together out of sub-components plus admin.

Large scale admin

Obviously large organizations want to do large scale admin of hundreds or thousands of machines and users. But that isn't the whole story. Vendor and third party support organizations also want to do this. What are the essentials for administering lots of systems?

- *Telepresence*[9]: You can do everything from one place over the network—no running around.
- *Program access*: You can do everything from a program—nothing has only a GUI. That way you can do the same thing many times with the same human effort as doing it once. Of course there have to be parameters, to accommodate variations.
- *Leverage*: You can say what you want declaratively, and software figures out how to get it done in many common cases. You can say each thing once, and have it propagated to all the places where it applies. To keep a number of machines in the desired state you describe that state once with a predicate and install that predicate by name on each machine. Then the system maintains each machine in the desired state automatically. You don't give a procedure for getting into the desired state, which probably won't work for some of the machines.
- *Problem reporting*: Currently it's an incredible pain to report a problem. All the "what was the state" and "what happened recently" should be automatic. For this to work the components obviously have tell you what to report.

Certification

A customer should be able to assemble a system (hardware and/or software) from approved components, or assemble a collection of components that passes a "this system is approved" test, and be pretty confident that it will just work. For the components, this means writing specs and testing for conformance with them. For the system, which won't really have a spec, it at least means testing that things hang together and don't conflict.

Of course there shouldn't be anything compulsory about certification. People could also live wild and free as they do today, without any such assurance. But perhaps we could do a bit better by pointing to the uncertified components that are causing the problems.

[9] I mean appropriate telepresence—we won't need videoconferencing any time soon.

Defining the state

The most important architectural idea after modularity is to pay careful attention to the system state. Here ‘state’ means state that is relevant to admin, stuff the users can change that is not their ‘content’. Some examples: file names and properties, style definitions, sort order in mail folders, network bindings. ‘State’ doesn’t mean the page table or the file allocation table; those are not part of the user model. Of course the line between admin state and content is not sharp, as some of the examples illustrate. But in general the admin state exists separately from the content.

Today the state is scattered around all over the place. A lot is in the registry in Windows, or in environment variables in Unix, though most of it is very application-specific, without common schemas or inheritance. But a lot is also in files, often encoded in a complicated way. There’s no way to dump out all the state uniformly, say into a database.

Microsoft Word is an interesting example. It stores state in the registry, in several kinds of templates, in documents, in files (for instance, the custom dictionaries) and in the running application. It has dozens of different kinds of state: text, a couple of kinds of character properties, paragraph, section, and document properties, fields, styles, a large assortment of options, macros, autotext and autocorrect, windows on the screen, and many more. It has several different organizing mechanisms: styles, templates, document properties, “file locations”. Some things can be grouped and named, others cannot. It’s not that any of this is bad, but it’s quite a jumble. There are lots of chances to get tangled up, and you don’t get much help in getting untangled.

Other things that are much simpler are much worse. I won’t try to catalog the confusing state of a typical mail client, or of network configuration. I don’t know enough to even try to write down the latter, although I know a lot about computer networking.

Design principles

The basic principle is that both users and programs should be able to easily see and change all the state. Again, ‘state’ here means state that is relevant to admin. That includes resource consumption and unusual conditions.

Second, the state should be understandable. That means that similar things should look similar, and related things should be grouped together. The obvious way to achieve this is to mak all the state available through a database interface so you can use database tools to view and change it, and the database methodology of schemas to understand it. This is good both for people and for programs. It’s especially good for customizing views of the state, since millions of people know how to do this in the context of database queries, views, reports, and forms. This has never been tried, and the state may be too messy to fit usefully into the relational mold.

This does not imply that the ‘true’ state is stored in a database; aside from compatibility, there are lots of reasons why that’s not a good idea. It’s pretty easy to convert from any other representation into a database representation. Reflecting database updates back into another representation is sometimes more difficult, but with careful design it’s possible. It’s not necessary to be able to change all of the database view. Some parts might be computed in such a way that changing them doesn’t make sense; this is just what we have today in databases.

It’s also a good idea to have an HTML interface, so that you can get to the state from a standard web browser. Perhaps the easiest way to get this is to convert to database form first and use an existing tool to make HTML from that.

Naming and data model

To have general admin, we need a uniform way to name all of the state and some standard ways to represent it externally. This is usually called a ‘data model’. Of course anything can be coded into any data model, but we want something natural that is easy both to code and to use.[10]

A hierarchical name space or tree is the obvious candidate, already used in file systems, object properties, the Windows registry, and the Internet’s SNMP. It’s good because it’s easy to extend it in a decentralized way, it’s easy to map most existing data structures into it, and it’s easy to merge two trees using a first-one-wins rule for each path name. The abstraction that goes along with trees is path names.

We studied hierarchical naming in handout 12. Recall that there are three basic primitives:

```
Lookup(d, pn) -> ((D, PN) + V)
Set(d, n, v)
Enum(d) -> SET N
```

With this `Lookup` you can shift responsibility for navigating the naming graph from the server to the client. An alternative to `Enum` is `Next(n) -> N`.

Directories can have different code for these primitives. This is essential for admin, since many entities are part of the name space, including file systems, DNS, adapters and i/o devices, and network routers.

Nodes may have types. For directory nodes, the type describes the possible children. In a database this information is called the ‘schema’. In SNMP it is the ‘management information block’ or MIB.

The other obvious candidate for a data model is tables, already used in databases, forms, and spreadsheets. It’s easy to treat a table as a special case of a tree, but the reverse is not true: many powerful operations on tables don’t work on trees because there isn’t enough structure. So it seems best to use a naming tree as the data model and subclass it with a table where appropriate.

Describing the good states: Predicates

The general way to describe a set of states too big to be enumerated is to write down a predicate (that is, a Boolean function) that is true exactly on those states. Since any part of the state can be named, the predicates can refer to any part of it, so they are completely general.

For this approach to be useful, we have to be able to state predicates formally so that the computer can process them. Some examples of interesting predicates (not stated formally):

> This subtree of the file system (for instance, the installed applications) is the same as that one over there (for instance, on the server).

[10] We want to stay out of the Turing tarpit, where everything is possible and nothing is easy (Alan Perlis).

All the .dll's referenced from this .exe are present on the search path, in compatible versions.

Tex is correctly installed.

All the connected file servers are responding promptly.

There is enough space for you to do the day's work.

All the Word and Powerpoint files have been converted to the latest version.

We also have to be able to process the predicates. Here are three useful things to do with a predicate that describes the good or desired states of a system:

- *Diagnose*. Point out things about the current state that stop it from being good, that is, from satisfying the predicate. For static state this can be a static operation. Active components should be monitored continuously for good state and expected performance; in most systems there are things that hang up pretty often.
- *Install or repair*. Change the state so that it satisfies the predicate. It's best if the needed changes can be computed automatically from the predicate, as they usually can for replication predicates ("This should be the same as that"). Sometimes we may need rules for changing the state, of the form "To achieve X, make change Y". These are hard to write and maintain, so it's best to have as few of them as possible.
- *Analyze*. Detect inconsistency (for instance, demands for two different versions of the same .dll), weirdness, or consequences of a change (will the predicate still be true after I do this?).
- *Anticipate* failures. Run diagnostics and look at event logs to give advice or take corrective action when things are abnormal.

Stating predicates

Predicates that are used for admin must be structured so that admin operations can be done automatically. If a predicate is written as a C function that returns 0 or 1, there's not much we can do with it except evaluate it. But if it's written in a more stylized way we can do lots of processing. A simple example: if a predicate is the conjunction of several named parts (Hardware working, Word installed, Data files backed up recently, ...) with named sub-parts, we can give a more precise diagnosis ("Word installation is bad because winwrd.hlp is corrupted"), find conflicts between the parts, and so on.

The stylized forms of predicates need to be common across the system, so that:

Users can learn a common language.

Every component builder doesn't have to invent a language.

The same predicates can work for multiple components. Example: "I don't want to lose more than 15 minutes of work." Applications, file systems, and clusters could all contribute to implementing this.

We need a common place to store the predicates, presumably the file system or the registry. This saves coding effort, makes it possible for multiple components to see them, and makes it easier to apply them to a larger environment.

We also need names for predicates, just as we need them for all state components and collections of values, so they can be shared, composed, and reported to the user in a meaningful way. Often giving a predicate a name is a way of reducing the size of the state. You say "Word is installed" rather than "winword.exe is in \msoffice\winword, hyph32.dll is in \msoffice\winword\, ...". Today many systems have weak approximations to named predicates: "profiles" or "templates". Unfortunately, they are badly documented and very clumsy to construct and review. They usually also lack any way to combine them, so as soon as you have two profiles it's a pain to change anything that they have in common.

What stylized forms for predicates are useful? Certainly we want to name predicates. We also want to combine them with 'and', 'or', 'not', 'all', and 'exists', as discussed earlier. The rest of this section discusses three other important forms. Predicates with parameters are essential for customizing. Predicates that describe replication and dependency are the standard cases for admin. They let you say "this should be like that except for ..." and "this needs that in order to work". Just evaluating them tells you in detail what's wrong: "this part of the state doesn't agree" and "this dependency isn't satisfied". Furthermore, it's usually obvious how to automatically make one of these predicates true. So you get both diagnosis and automatic repair.

Customizing: Predicates with parameters

Global names are good because they provide a meaningful vocabulary that lets you talk more abstractly about the system instead of reciting all the details every time. Put another way, they reduce the size and complexity of the state you have to deal with. Parameters, which are local names that you can rebind conveniently, are good because they make the abstractions much more general, so each one can cover many more cases: "All the files with extension *e* have been backed up", or "All the files that match filter *f*", instead of just "All the files". Subroutines with parameters have the same advantages that they do for programming.

Computed values are closely related to parameters, since computing on the parameters makes them much more useful. The most elaborate example of this that I know of is Microsoft Word templates, which let you compute the current set of macros, menus, etc. by merging several named templates. The next section on replication gives more examples.

Replication

This is the most important form of predicate: "Mine should be like hers, except for ...". In its simplest form it says "Subtree A (folder A and its contents) is the same as subtree B". Make this true by copying B over A (assuming B can't be changed), or by reconciling them in some other way. Disconnected operation, caching, backup, and software distribution are all forms of replication, as well as the obvious ones of disk mirroring and replicated storage servers.

Variations on replication predicates are:

> "A contains B, that is, every name in B has the same value in A, but A may have other names as well." To make this true, copy everything in B over A, leaving things that are only in A untouched.
>
> "A is ahead of B, that is, every name in B is also in A with a version at least as recent." To make this true, copy everything in B over a missing or earlier thing in A.

All of these are good because they can control a lot of state very clearly and concisely. For example, a department has a predicate "standard app installation" that says "Lotus Notes is installed, budget rollup templates are installed, ...", and "Lotus Notes is installed" says "C:\Lotus\Notes equals \\DeptServer\Notes, ...", etc.. Then just asserting "standard app installation" on a machine ensures that the machine acquires the standard installation and keeps it up to date.

In all these cases B might be some *view* of a folder tree rather than the actual tree, for instance, all the items tagged as being in the minimal installation, or all the *.doc files. (I suppose A might be a view too, though I can't think of a good example.) Or B might contain hashes of values or UID's rather than the values themselves; this is useful for a cheap check that A is in a good state, though it's obviously not enough for repairing problems.

Replication predicates are the basic ones you need for installing software; note that there are usually several A's: the app's directory, a directory for dll's, the registry. Replication predicates are also what you need for disconnected operation, for backup, and for many other admin operations.

See the later section on coding techniques for ways of using these predicates efficiently.

Dependency

This is the next most important form of predicate: "I need ... in order to work". Its general form is "If A, then there must be a B such that P is true." Dependency is the price of modularity: you can't just replicate the whole system, so you need to know how to deal with the boundaries. Programs have dependencies on fonts, .dll's, other programs, help files, etc. Documents also have dependencies, on apps that interpret them, templates, fonts, linked documents, etc.. Often unsatisfied dependencies are the most difficult problems to track down.

The hardest part about dependencies is finding out what they are. There are three ways to do this:

- Declare them manually (I need this version of `foo.dll`; I need COM interface `baz`; I need Acrobat 4.0 or later).
- Deduce them by static analysis (What are all the links from this document, what fonts does it use, what Corba components? What .dll's does this .exe use statically?).
- Execute and trace what gets used.

These approaches are complementary, not competitive. In particular, a static analysis or a trace can generate a declaration, or flag non-compliance with an existing declaration.

Given dependencies, we can do a lot of useful analysis.

> Are all the dependencies satisfied now?
>
> What will break if I make this change?
>
> Do the requirements of two apps conflict?
>
> How can this dependency be eliminated?
>
> When something goes wrong, what state needs to be captured in reporting the problem?

Components should provide operations for getting rid of dependencies, by doing format conversions, embedding external things like fonts and linked documents, etc. There is an obvious tradeoff between performance and robustness here; it should be under control of the admin policies, not of obscure check boxes in Save dialogs.

The key to fast startup (of the system or of an app) is pre-binding of facts about the environment. The problem is that this creates more dependencies: when the environment changes the pre-bindings become obsolete, and checking for this has to be fast. If these dependencies are public then the system can check any changes against them (probably by processing the change log with a background agent) and notify the app next time it runs. Otherwise the app can scan the change log when it starts up. A problem is that it's hard to register these dependencies reliably; it may have to be done by tracing the component's calls on other components.

Resource allocation

Resource allocation is an aspect of the state that is sometimes important for workstations and always important for shared servers. We need predicates that describe how to allocate storage, CPU cycles, RAM, bandwidth, etc. among competing apps, users, documents, or other clients. 'Committed' resources like disk storage or modems can't be taken away and given back later; they need special treatment. This is why HSM (hierarchical storage management) is good: it makes disk a revocable rather than a committed resource, as VM does for RAM.

A related issue is garbage collection. Abstractly, this follows from predicates: any state not needed to satisfy the predicates can be discarded. It's unclear whether we can make this idea practical.

Coding resource allocation in a system with more than one component that can do a given task requires load balancing. It also requires monitoring the performance, to detect violations or potential violations of the predicates. Often the predicate will be defined in terms of some model of the system which describes how it ought to respond to an offered load. Monitoring consists of collecting information, comparing it to the model, and taking some action when reality is out of step. Ideally the action is some automatic correction, but it might just be a trouble report to an administrator.

One important form of monitoring is keeping track of failures. As we saw in handout 28, a fault-tolerant system tends to become fault-intolerant if failed component are not repaired promptly.

Coding techniques

The basic coding strategy is to start with the "real" state that a component runs against, and compute a view of the state that's easy to present and program. This way admin isn't tied down by legacy formats. The computed view is a kind of cache, with the same coherence issues. Of course the idea works in the other direction too: compute the stuff a component needs from stuff that's convenient for admin.

To make this efficient, if a component has a state that's expensive to read, it writes a change log. Entries in the log pinpoint the parts of the state that have changed. Admin reads this log plus a little of the state and tracks the changes to keep its view coherent with the real thing. The file system, for instance, logs the names or ids of files that are written or renamed. Note that this is much simpler and more compact than a database undo/redo log, which has to have all the information needed to undo or redo changes. Some components already generate this information in the form of notifications, but recording it permanently means that apps that are not running at the time can still track the state. Also, writing a log entry is usually much cheaper than calling another program.

Change logs with some extra entries are also good for monitoring performance, for diagnosis, and for load balancing.

There may be no change log, or it may be incomplete or corrupt. In this case a way to pinpoint the parts of the state that have changed is to remember hashes of the files that represent it. Recomputing a hash is fast, and it's only necessary to look at a file if its hash is different. This technique can be applied recursively by hashing folders as well (including the hashes of their contents).

A more complete log makes it possible to undo changes. This log can take up a lot of space if a lot of data is overwritten or deleted, but disk space is often cheap enough that it's a good deal to save enough information to undo several days worth of changes.

Words of wisdom from Phil Neches (founder of Teradata)

1. It's cheaper to replace software than to change it.
2. It's cheaper to process, store, or communicate than to display
3. It's cheaper to be networked than standalone. The implications for software development are now widely accepted: continuous updates, shared data, and availability through replication.
4. Public transactions are cheaper than anonymous ones. This is because of accountability. For example, credit cards are cheaper than cash (after all costs are taken into account).

Finally, software has its face to the user and its back to the wall.