Automated Client-side Integration of Distributed Application Servers

Conrad E. Kimball, Vincent D. Skahan, Jr., David J. Kasik – The Boeing Company
Roger. L. Droz – Analysts International

ABSTRACT

From the Single Glass Program Plan, dated August 18, 1997:

Vision: Provide BCAG users access to all applications and data needed to perform their respective jobs from a single desktop environment with acceptable levels of function, performance, and reliability.

The Single Glass program in the Boeing Commercial Airplane Group (BCAG) provides Engineering UNIX users the ability to access all required UNIX and PC applications and associated data via a standardized Common Desktop Environment (CDE) desktop.

The goal is to do this with enough performance, reliability, and transparency so that each engineer only needs one computer (i.e., a "Single piece of Glass") on their desk, thereby reducing the number of desktop devices required per engineer. There is an additional process benefit by increasing the amount of concurrent design and analysis possible.

The Single Glass desktop provides unified access to over 350 locally executed applications previously provided in a number of separate legacy environments from the shell and also within a common CDE look-and-feel developed through formal usability studies done in Seattle and Wichita. Currently running on over 5000 IBM RS6000 workstations worldwide for over 6000 users, Single Glass is designed to support IBM, HP, Sun, and SGI UNIX and NT workstations.

This paper describes some of the design decisions and project constraints that led Single Glass to decide to deliver a unified logical namespace to the clients that spans multiple physical servers. This was accomplished by automating creation of the required symbolic links to the many distributed file servers rather than simply building one monolithic "union of all the supplier trees".

This implementation has been proven to work consistently on AIX, HP-UX, Solaris, and Linux platforms.

Name Space

Reliable delivery of applications that have been developed in multiple organizations is a key Single Glass problem.

Boeing-written applications are organized via a standardized Boeing Common Directory Structure (CDS) analogous to many of the various file system naming standards in place elsewhere. A similar naming convention is in place for commercial-off-the-shelf (COTS) software.

The intent of CDS is to standardize the Boeing internal name space presented to the users, and to provide guidance to the software developers regarding where (and how) to install their software. Single Glass considers CDS to be the combination of public and private areas under one /boeing namespace [Figure 1].

In general, CDS requires the public namespace to consist solely of symbolic links to software components in the private namespace of that component's supplier. In addition, there are common directories to contain application configuration and log files.

The actual files that implement an application are encapsulated in a sub-tree of arbitrary complexity under /boeing/sw/<product>. Applications expose their public interface by installing symbolic links in known directories such as /boeing/bin, /boeing/man, and /boeing/lib.

Users include the stable set of public directories in $PATH, $MANPATH, and the like, and the symbolic links present there take care of providing access to the default (or specified) version of the application they wish to run from within the "private" portion of the /boeing tree.

A common script automates installing applications into the private namespace in the /boeing tree and modifying the appropriate public links so the "stable public path" delivers the current default version of the applications to the user.

Typical Software Installation

Figure 2 shows a minimal installation of version a02 of a hypothetical product "foo" consisting of one executable with a corresponding manual page.

The common installer tool installs the actual software in the private directory of the application supplier, and creates the appropriate links in the public directories which are referenced in every user's $PATH and $MANPATH.

Both version dependent and independent links are present in the public directories under /boeing to permit the users to run either the default version or a user-specified version of each product. The products are compiled and configured to use the version-dependent paths so that multiple versions of any product may be present at the same time. Users reference the "stable" path /boeing/bin/foo for the current default version of the product, or can choose to reference an absolute version-specific path of /boeing/bin/foo_a02 to get to version a02 of the product (for example).

Similarly, they can view the manual page for the product by specifying the default product name "man foo" or the version-specific name "man foo_a02".

The installer always uses fully qualified (rather than relative) pathnames when creating symbolic links in order to permit the physical implementation of a /boeing tree to span multiple filesystems (glued with automounter or the like) if necessary.

Multiple Fileservers

Rather than split a single /boeing tree onto multiple fileservers, Single Glass does essentially the opposite. We automate assembly of a single logical tree on the client side from multiple whole application fileserver trees provided by a number of organizations within Boeing [Figure 3].

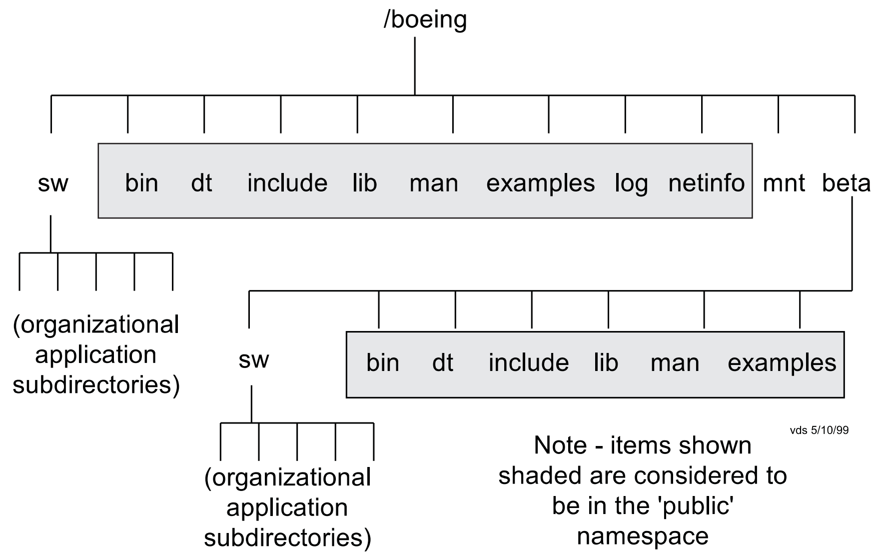

Figure 1: Boeing common directory structure (excerpt).

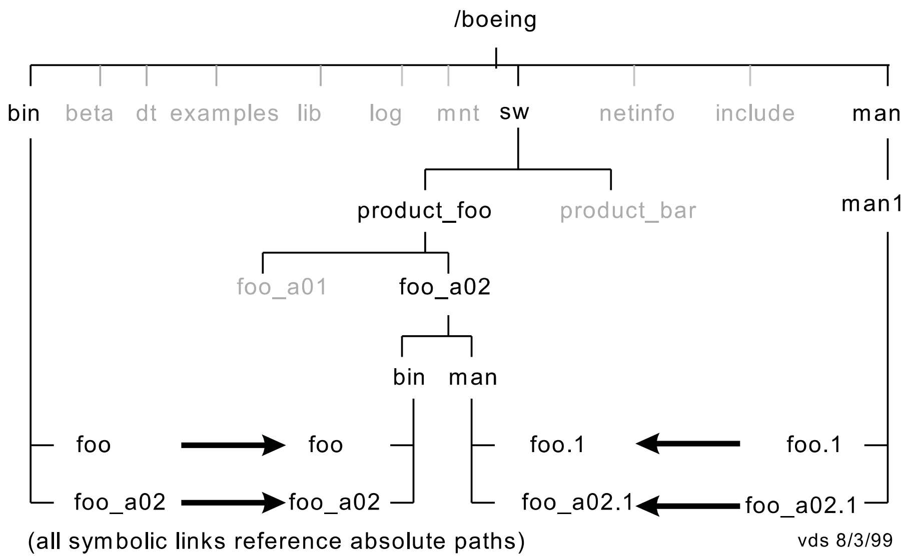

Figure 2: Typical product installation in common directory structure.

Single Glass currently supports applications coming from at least seven different application providers, many of whom supply their own fileserver capacity to be mounted by the clients. There is no (current) effective way to pool these servers (and budgets), nor is this considered wise strategically as scale increases.

Traditionally, the Single Glass project would have assembled a monolithic union of the various application provider unique /boeing trees and presented that unified tree to the clients through one project-wide /boeing mount of that assembled tree. This was impractical for a number of reasons.

- There are limits related to "how big a fileserver is possible or wise".
- Our implementation is being done in a staged manner over a long period of time rather than as a "big bang" event, and procurement of interim fileservers to provide sufficient capacity during the transition period of over 24 months was impractical.
- The very act of performing an integration into one physical /boeing tree structure tends to induce territorial and process conflict among the contributing application communities with respect to managing application content and versioning.
- We assume a heterogeneous audience that uses multiple versions of the same application. Our distributed user community has diverse views

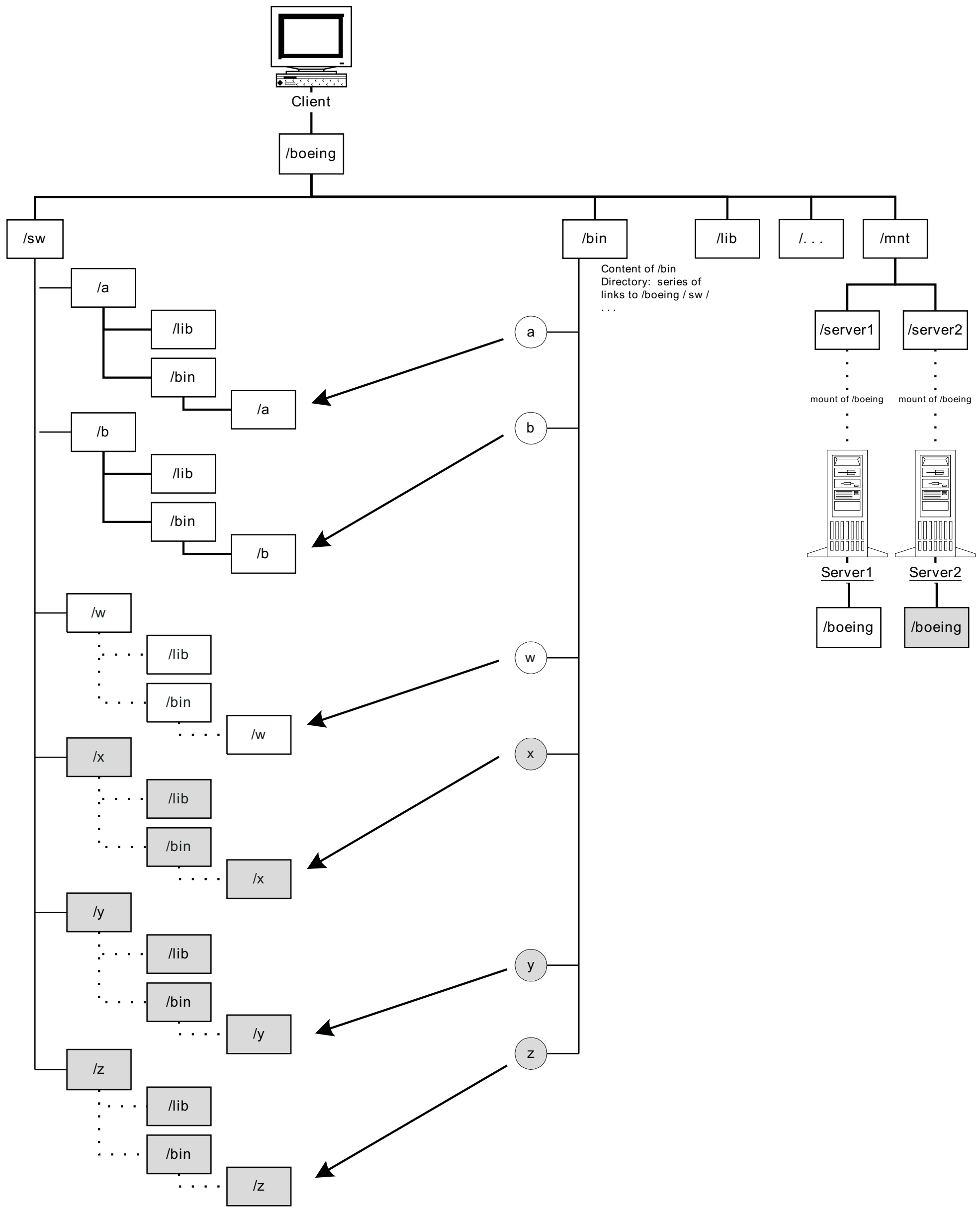

Figure 3: Client-side view of application servers.

regarding the need to standardize on one version of each product or not, and agreeing on a single version of any application is difficult to implement.

- Our application suppliers are used to a great deal of autonomy in how they develop, test, and deliver their software. Each supplier tends to have their own unique installer tool that installs the software into /boeing in a CDS compliant manner. This autonomy led us to a solution where we would need to assemble an integrated union of multiple independently "valid" supplier-provided /boeing trees.
- The requirement to support the legacy Boeing Common Directory Structure (which mandates extensive use of symbolic links) made a simple automounter solution impossible.
  Automounter would have been a fine solution for assembling the private portions of the /boeing tree (each of the subdirectories under /boeing/sw) but it cannot assemble the extensive sets of symbolic links in the public portions of the tree.
  By creating a local composite /boeing/bin on the client, we also tend to avoid multiple network transits by having a local /boeing/bin in $PATH rather than multiple NFS mounted directories.
- Our design allows locally installed /boeing tree software to blend with /boeing tree software from multiple NFS servers. This required the assembly of the /boeing tree to be moved to the client-side so that each client can have its own unique local content if required.

Client-side assembly minimizes disk space and software distribution headaches involved with maintaining the entire /boeing tree locally on the client system, while permitting some clients to have local installations if required. It also provides some caching effect as all the items in $PATH, $MANPATH, and the like are resident locally as symbolic links.

Accordingly, the decision was made to perform client-side assembly of whole /boeing trees with automated tools.

Automating the Client-side Assembly Process

Single Glass builds a composite client-site /boeing tree consisting of (generally minimal) client-side local content and many symbolic links pointing to the various remote servers mounted under /boeing/mnt.

This is done on a first come, first served basis (alphabetically), under a local /boeing/mnt, with local content having the highest precedence [Figure 4].

The easiest way to understand this process is to imagine laying several CDS trees on top of each other. Ideally, the contributing trees will merge with no conflict. In practice, when conflicts do occur, they are resolved in favor of the first contributor. The local workstation takes highest priority, followed by the servers as they appear in alphabetical order in the /boeing/mnt directory.

The various /boeing trees are dynamically "layered" on a first-come-first-served basis by a Single Glass-provided perl program that "integrates" the multiple server trees into a consistent client-side view.

This process takes between 90-180 seconds and creates approximately 5200 required symbolic links on the client-side. Virtually all the actual time is spent calculating what directories and symbolic links to add, delete, or modify locally on the client.

Since server-side changes are (hopefully) relatively infrequent, the integration script is only run once nightly via cron on all workstations or at the end of the system boot sequence. In this way, a simple reboot will restore the client to "last known-good" configuration.

The integration script adds, deletes, and modifies the appropriate links on the client so the workstation can reference the combination of locally installed software and software residing on several code servers through a single unified /boeing.

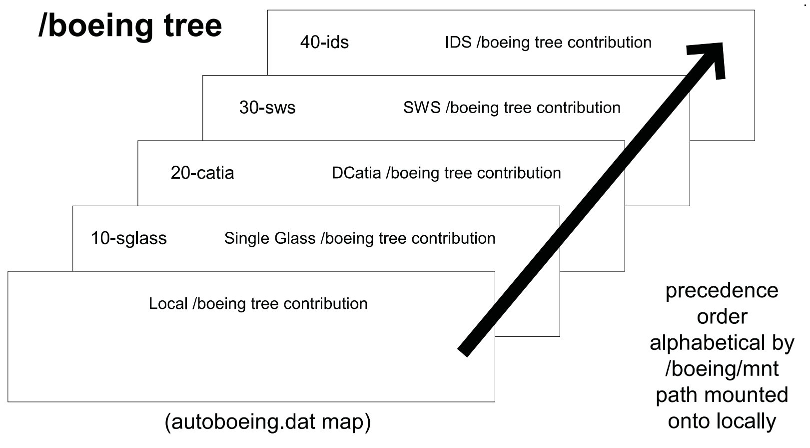

Figure 4: Layered application servers.

The integration script:

- Reads the master automounter map and identifies all trees that could be (auto)mounted under /boeing/mnt.
- Reads a pre-computed inventory file (a compressed "ls -AlRc") from each of those servers for efficiency reasons.
  If no inventory file is available, it walks the tree to calculate an inventory, with considerable network and server load due to the size of the /boeing trees.
- Compares the local /boeing tree with the union of the remote tree inventories.
- Adds, deletes, modifies the appropriate links and directories to create the combined union of all the trees on the local client /boeing filesystem.
- Cleans up any obsolete links, removes any local /boeing tree subdirectories that are empty.

This assembly process takes place at the individual workstation level, so that all users of a particular workstation have an identical view of the available applications.

Similarly, since workstations use the same automounter maps, each workstation in a particular NIS domain is integrated identically, yielding identical client-side configurations on each workstation.

Users can of course choose to alter their personal defaults by customizing (at their own risk) their personal dot files. In case the user's customizations make their account unusable or unstable, we provide a CDE action to quickly move their modifications aside and reset the account to a known good initial configuration.

How the Script Works Internally

CDS is organized around "products" and "portfolios", which are essentially collections of products developed by one organization.

The integration script first identifies and builds symbolic links to all "products" and "portfolios" located under /boeing/sw [Figure 5 – item 1]. The client can now see the "private" or "physical" installation tree of all software packages in the network.

Next, the script merges the "public interface directories." The corresponding directories are searched on each contributing tree. Links that point into the private directory structure of a "product" contributed by a given server are reproduced in the client's public interface directory [Figure 5 – item 2].

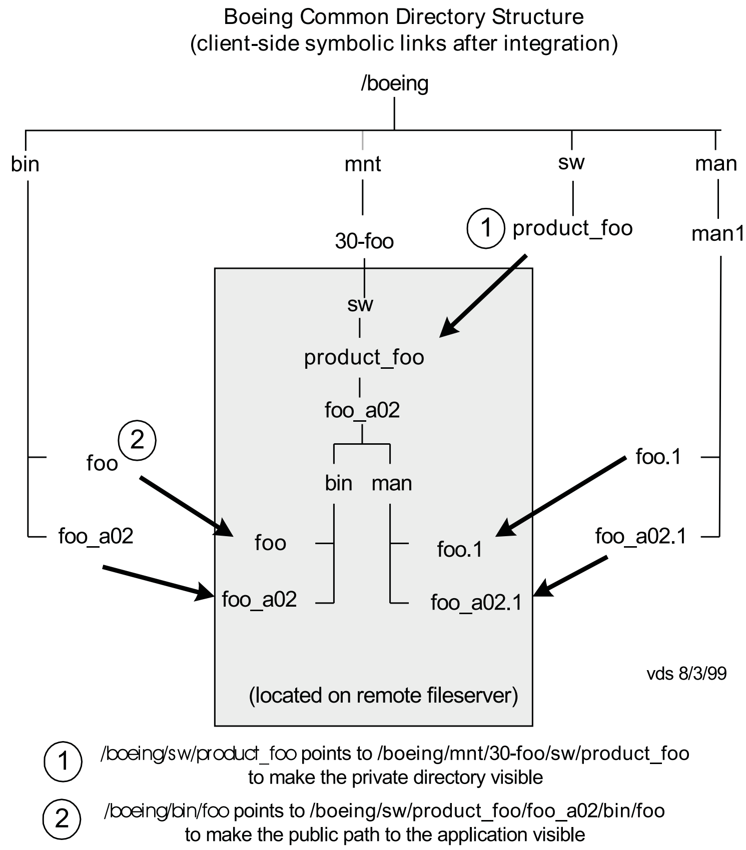

Figure 5: Client-side view after integration.

The script recursively descends the public interface, reproducing links to files, until a link to a directory in the private namespace is encountered.

Last, the script, if requested, will delete local links that did not come from one of the contributing /boeing tree servers unless they meet one of the following conditions:

- are located in /boeing/netinfo on the local workstation
- are located in /boeing locally, and also listed in the /boeing/netinfo/preserve.aix file (so called "precious" files as mentioned in the actual perl code)

When the client is unable to integrate a particular tree due to a remote server-down situation, the pre-existing links from that server are assumed to still be appropriate and are left intact. Links pointing to a server that returns a permission-denied message on the automounter attempt to mount are considered no longer valid and are deleted.

In addition, the integration script will log, but not create, any links that are calculated to be "dangling", so that the client-side configuration only reflects applications that will (presumably) be functional after the integration is complete.

STDOUT and STDERR are captured to log files in /tmp on the local workstation that can be used for debugging purposes.

It is important to note that the integration script is indeed an integrator, not an installer. It blindly believes that the application provider has ensured that their software installs so that the public/private directory separation is in place, and equally blindly copies the link values present in those remote trees to the client side.

While the integration script has some knowledge of the requirements of the CDS standard and can log discrepancies it finds, it has no knowledge of what the link values in the public directories should be. That is the responsibility of the application provider's installer tool.

Caching Server Inventory To Reduce Clock Time And Network Load

Early in the integration phase of Single Glass, we identified that there can be a tremendous amount of traffic generated to inventory a large application server once, let alone 5000 times in parallel.

Pre-computing a compressed inventory of the fileservers, and having the clients "quickly read a remote file then work totally locally" mitigates this cost.

The inventory script only writes a new inventory file into place if there have been changes made that the integrate script "cares about". This permits the served inventory files to be used as timestamp sentinels for comparison to permit "lazy (i.e., do it only if needed) integration" client-side.

This pre-computed inventory eliminated over 99% of the network load and 50% of the clock time needed to synchronize the Single Glass clients with the /boeing tree servers (vs. our initial implementation).

Total NFS load for a full integration is currently approximately 1300 NFS operations as follows:

Nfsstat(1M) call	Number of Calls
Getattr	27
Lookup	501
Access	75
Readlink	2
Read	671
Readdir+	11
Fsinfo	5

Integration takes 90-180 seconds depending on client CPU speed (IBM model 42T and 43P workstations).

Randomized "Lazy" Integration

In all cases, the integration script only integrates the clients when there are changes to be made (so-called "lazy" integration).

This is established by examining the timestamps on the local log files versus the inventory files on each of the servers.

If any of the remote server inventories is newer than the local log file, the client does a full integration to synchronize itself to the union of the remote server configurations.

Since the integration script is called from an identical crontab entry on 5000 clients that have synchronized clocks, a client-side ksh script and accompanying C program serve as a randomizer to ensure that all the systems don't "wake up at once" and overload the file servers or networks. Both files are stored locally in on the workstation to minimize network load and are the only local content required to make the integration occur.

The compiled program determines, based on the TCP address of the local system, how many seconds to sleep before calling the integration script with the proper options.

The current settings are intended to have all systems sleep for no more than one hour before waking up and doing the actual /boeing tree integration, but this value is also configurable at run-time if needed.

Lastly, client-side integration can be "forced" to supersede a calculated answer of "no integration required".

Ensuring The Clients Are Up To Date After Reboot

Frequently workstations are relocated, or are booted from alternate drives for special beta testing and the like, and thus miss their once-per day scheduled cron-based integration. To ensure that the systems are always in the most up-to-date possible state after a reboot, the integrate script is called out of the client boot sequence after automounter is up.

Non-technical Challenges Of Client-side Integration

There are of course downsides to client-side integration that tend to be more cultural (Fear-Uncertainty-Doubt) than technical in nature.

- There is the need to synchronize multiple server installations across many sites (but of course doing so on 25 fileservers is better than doing so on 5000 clients).
- The traditional approach to providing content consisting of a large number of applications is to provide an equally large file server rather than our distributed logical approach.

  Management, especially when migrating from a mainframe model, often questions the reliability and availability of such a distributed approach.

  Fortunately, recent problem report call analysis shows average measurable downtime of less than one unscheduled hour per month per workstation.
- Our distributed customer base has widely varying definitions of “acceptable risk”, “required testing”, and ability to be the recipient of change in their computing environments.

  Our distributed logical file server model permits application versions to be updated asynchronously to each other at each application user community’s own pace. This causes certain “culture collisions” among the different user communities regarding how many combinations of versions need to be tested against each other.
- Client-side integration introduces one more (final) step in the client workstation build/bringup procedure. This has required some changes to the workstation system assembly procedures and training. Conclusions

The architecture and tools described above have permitted us to logically layer a number of existing large application fileserver implementations into a consistent client-side view of the union of the possible applications available for use on over 5000 workstations worldwide.

This has been done within a well-defined common directory structure and associated delivery system implementation constraints and rules. Our approach has succeeded in not affecting the implementation of the legacy systems or requiring change in the existing software installation tools used by the internal software developers.

Rather than increasing total project cost, this implementation has lowered the cost of entry into Single Glass for a new application provider, as we can avoid the costs (and potential battles) of requiring potential application providers to be “assimilated” into the project.

The result of our implementation is an architecture that has proven to be portable, extensible, reliable, and supportable worldwide at a 5000 workstation scale.

Futures

Given our heavy reliance on NFS, we need to investigate NFS caching to improve performance and minimize traffic.

Our cached inventory files on the fileservers contain quite a bit more information than the integrate script actually needs. We believe shrinking the precomputed inventories to the minimum needed for the integrate script to do the job would result in load and time improvements.

While the various links created by the integration script are resident client-side, CDE reads through the symbolic links at user login time when building the application manager desktop, causing a several second delay in logging in as each fileserver providing pieces of the assembled /boeing/dt is automounted. Bringing the CDE tree assembly process fully client-side by copying the CDE related files (rather than just links) into the more traditional /etc/dt directory during the integration process on the workstation should speed up login time noticeably.

The users always expect their systems to be always “up” and always “up-to-date”. In a worldwide 24x7x365 company, this is a major long-term issue to overcome.

Any fileserver-side changes do not take effect until the server is re-inventoried, and the client is re-integrated (which normally happens just on reboot, or nightly via cron). Re-integrating the client while a user is logged in can have adverse affects to the user’s session (i.e., changing the default version of “foo” while they have a session of the old version running).

We need to be able to have the clients check “should I catch up” more often (or be notified automatically) and have them integrate “on the fly” without affecting any logged in users or running jobs in any way.

Author Information

Conrad Kimball <Conrad.Kimball@boeing.com> is an Associate Technical Fellow of the Boeing Company. He is a co-architect for the Puget Sound Single Glass program, and is a member of the Boeing-wide computing delivery system Technical Leadership Team and Technical Planning Board. Conrad holds a Master of Software Engineering degree from Seattle University.

Vince Skahan <Vince.Skahan@boeing.com> is a System Design & Integration Specialist at the Boeing

Company. He holds a B.S. in Chemical Engineering from Drexel University, and has been doing large scale heterogeneous unix administration since 1987.

David J. Kasik <David.J.Kasik@boeing.com> is a Technical Fellow of the Boeing Company. He is a co-architect for the Puget Sound Single Glass program and acts as the Geometry and Visualization architect for Boeing Commercial Airplanes.

Roger Droz' <Roger.Droz@seaslug.org> career has spanned a broad range of computing environments as a designer and programmer – from 8 bit embedded systems to the enterprise scale of Single Glass; from operating system kernel and device drivers to scientific and commercial applications. Roger holds a Masters degree in Electrical Engineering from Washington State University.

References

Linux filesystem standard online at http://www.pathname.com/fhs/ .

Common Directory Structure and Supporting Environments for BCAG Unix Systems, Boeing internal document D6-81580 dated 3/10/95.

Kasik, D., Kimball, C., Felt, J., Frazier, K. *A Flexible Approach to Alliances of Complex Applications*, presented at ICSE'99, http://sunset.usc.edu/icse99.html .

Bell, John D., *A Simple Caching File System for Application Serving*, http://www.usenix.org/publications/library/proceedings/lisa96/full_papers/jbell4.html .

Hauser, C., *Speeding Up UNIX Login by Caching the Initial Environment*, http://www.usenix.org/publications/library/proceedings/lisa94/hauser.html .

Ph. Defert, E. Fernandez, M. Goossens, O. Le Moigne, A. Peyrat, I. Reguero, *Managing and Distributing Application Software*, http://www.usenix.org/publications/library/proceedings/lisa96/full_papers/reguero/reguero.txt .

Furlani, John L., Osel, Peter W., *Abstract Yourself With Modules*, http://www.usenix.org/publications/library/proceedings/lisa96/pwo.html .

Wong, Walter C., *Local Disk Depot – Customizing the Software Environment*, http://www.usenix.org/publications/library/proceedings/lisa93/wong.html .