The Seven Deadly Sins of Distributed Systems

Steve Muir
Department of Computer Science
Princeton University

Abstract

Applications developed for large-scale heterogeneous environments must address a number of challenges not faced in other networked computer systems such as LAN clusters. We describe some of the problems faced in implementing the PlanetLab Node Manager application and present general guidelines for application developers derived from those problems.

1 Introduction

Developers of distributed applications face a number of challenges particular to their environment, whether it be a homogeneous cluster or a global environment like PlanetLab [6, 2], such as unreliable communication and inconsistent state across the distributed system. Furthermore, personal experiences developing and running applications on workstations and/or servers with LAN interconnects often are not applicable, particularly to problems such as resource allocation and sharing. We present a set of guidelines created in response to our experiences maintaining an infrastructure application for PlanetLab, a large-scale heterogeneous environment.

This paper describes the *seven deadly sins of distributed systems*—seven challenges we encountered in developing *Node Manager*, a key infrastructure component of the PlanetLab environment. Although these challenges are described in the context of a particular application we believe the underlying problems are common to all large-scale heterogeneous environments and thus of interest to a wider audience than just current and future users of PlanetLab.

In Section 2 we briefly describe the PlanetLab environment and Node Manager application in order to set the stage for Section 3's enumeration of the seven deadly sins. After describing each challenge, along with concrete examples of how Node Manager addresses it, we conclude with a summary of general principles applicable to the design and development of distributed applications. Note that this paper focuses on challenges faced by application developers, and so does not directly address the problems faced in administering distributed systems such as PlanetLab e.g., system security, effect of network traffic on external sites.

2 PlanetLab

PlanetLab is a global distributed environment consisting of 400+ nodes connected to the Internet at over 175 geographic locations, a mix of academic institutions and research laboratories. Its global nature provides an extremely diverse network environment that makes implementing a distributed application somewhat challenging.

PlanetLab nodes are IA-32 systems, usually running Intel Xeon or Pentium 4 CPUs with some Pentium III and AMD systems thrown into the mix. Each is centrally administered and configured using a bootable CD that ensures the node boots in a standard manner and can also be rebooted into a minimal *debug* mode; this boot mechanism is not intended to be secure against adversaries with local access but merely to provide a reasonably degree of assurance that nodes can be brought into a known state remotely.

The PlanetLab OS is derived from RedHat Linux 9 with a small number of kernel patches applied; the most significant patches are the standard *vserver* patches to provide namespace isolation between users, and *plkmod*, a kernel module that provides CPU scheduling and network virtualisation. These two components together provide users with an environment that appears essentially to be a complete Linux system, with a restricted root account used to configure that system by each user.

2.1 Node Manager

Node Manager (NM) is the infrastructure subsystem that configures the appropriate user environments—*slivers*—on each node. A user creates a *slice* in the PlanetLab Central (PLC) database, specifying such properties as nodes to be instantiated upon, authorised users and resources allocated to that slice. The contents of that database are

then exported to Node Manager as an XML file which is subsequently used to configure slivers on each node.

Node Manager provides users with prompt creation and deletion of slivers while contending with a large number of problems inherent to the distributed nature of PlanetLab. Our experiences developing and maintaining Node Manager have led us to create a list of the most important challenges to be addressed: we refer to these as the seven deadly sins of distributed systems.

2.2 Related Work

PlanetLab is neither the first nor only distributed environment available to network experimenters and developers of new services but we believe it to be both larger and more heterogeneous than earlier such projects. The University of Utah's *Emulab* [8] project provides a large cluster environment for researchers but lacks the heterogeneous network environment of PlanetLab; of course, for some applications that may actually be preferable. MIT's *Resilient Overlay Network (RON)* [1] testbed provides a similar network environment to PlanetLab but on a much smaller scale. Finally, the *Google file system* [3], a cluster-based filesystem, addresses many similar challenges to those encountered in PlanetLab even though the environment is much more homogeneous.

3 The Seven Deadly Sins

For each of the 'sins' listed below we will describe the nature of the problem along with the solutions adopted in Node Manager. While many of these problems are well-known and in some cases the subjects of vast amounts of existing research, we believe that the practical context in which we encountered them helps motivate the applicability of their solutions to PlanetLab and similar distributed systems.

1. Networks are unreliable in the worst possible way
2. DNS does not make for a good naming system
3. Local clocks are inaccurate and unreliable
4. Large-scale systems always have inconsistencies
5. Improbable events occur frequently in large systems
6. Overutilisation is the steady-state condition
7. Limited system transparency hampers debugging

Note that this list is far from complete, but reflects the set of challenges we faced implementing Node Manager, particularly where the problem or its effects were surprising.

3.1 Large Heterogeneous Networks are Fundamentally Unreliable

The single biggest challenge facing architects of distributed applications is the simple fact that networks are unreliable. Although the specifications of IP, TCP, UDP, etc., all make clear the various forms this unreliability can take, it is easy for application developers using high-level languages and systems to forget, or fail to appreciate, the practical implications of these problems for their applications. Often, this unreliability does not take a convenient form from the architect's perspective—the fate of network packets is not simply sent or discarded, they may also be delayed, duplicated and/or corrupted. Even 'reliable' protocols such as TCP still leave the designer to deal with such problems as highly variable latency and bandwidth and unexpected connection termination.

The implications of these problems for distributed application development are significant. Some are obvious—what should the application do if a connection is refused—whereas others are more subtle—what should the application do if a small file takes 24 hours to download, or if a remote procedure call (RPC) is interrupted?

Perhaps the foremost rule in handling network problems is that all possible errors should be handled gracefully—as the 5th problem described below says, it's the error condition that isn't handled that will be the one that occurs sooner or later when an application runs 24/7 on hundreds of nodes. Large, geographically distributed networks *will* exhibit all kinds of network failure modes, including the obscure ones that never occur on local networks: packet reordering, duplicate packets arriving at a destination.

Secondly, many transient failures occurring within the network, e.g., packet loss, are successfully hidden from applications by the network stack when in practice the application needs to be aware of them. For example, we frequently found that poor connectivity to our central servers would lead to file downloads, even of small files, taking many hours. This led to two problems: first, the application becomes tied up in file download, thus being unable to perform other operations or respond to new service requests; and second, the received data is often stale by the time download is completed. The first problem can be addressed in several ways, but we use two complementary solutions in Node Manager: multithreading (or asynchronous I/O) is used to perform long latency operations while responding to other events, and timeouts (if chosen appropriately) are effective in terminating operations that do not complete promptly. Stale data can be identified using timestamps, although validating such a timestamp has its own challenges (see Section 3.3).

RPC operations that terminate abnormally present another class of problem. The client typically has little or no information about the state that the server was in when

the operation terminated, so determining how to handle this problem depends to a great deal upon the nature of the operation. Transaction processing mechanisms are one solution but may be too heavyweight for the rapid prototyping of new network services that is the focus of PlanetLab; one solution adopted in Node Manager is for the server to maintain only minimal internal consistency properties such that an aborted operation does not adversely impact subsequent requests, and then have external applications periodically verify that the server state is globally consistent.

For example, Node Manager provides a pair of operations—*acquire* and *bind*—that are used together to acquire a set of resources, returning a handle to the caller, then use that handle to bind the resource set to a particular slice. If the acquire operation fails abnormally i.e., due to an RPC mechanism error rather than an error returned by Node Manager, the client has no information about whether the resource set was allocated. Since the client did not receive the resource set handle it has no way of determining that information, so it retries the request; in Node Manager itself we periodically cleanup allocated resource sets that have not been bound.

Finally, distributed applications may have to deal with interference from other network users. Random connections from external networks, such as port-scanning of network interfaces, are not uncommon, so applications providing network services must be able to handle such invalid connection attempts gracefully. In particular, higher-level libraries may have internal error handling designed for simple client-server applications that is inappropriate in the target environment.

3.2 DNS Names Make Poor Node Identifiers

A key challenge in any distributed system is naming of entities in the system [7]. We focus only on the challenge of naming nodes in the system i.e., assigning a name N to a node such that N unambiguously refers to that node and that node only, and the node can reliably determine, in the face of unreliable communication, that its name is N.

DNS names are appealing candidates as a solution for several reasons: they're simple, they're well understood by developers and users, and there's a large amount of infrastructure and tools to support use of them. Unfortunately, DNS names suffer from both *ambiguity* and *instability*—DNS names may not be unique and are not stable over long periods of time. These problems arise due to several root causes:

- Human errors: DNS names are assigned by system administrators, so there are often mistakes made: the same name assigned to multiple nodes or vice versa, reverse lookups not matching forward lookups.
- Network reorganisation: sometimes sites change their internal network addressing, thus requiring that DNS records be updated; hostnames are even occasionally changed to more 'user-friendly' variants e.g., `nwu.edu` changed to `northwestern.edu`.
- Infrastructure failures: DNS servers may fail completely or be overloaded, thus forcing requests to be sent to a secondary server.
- Network asymmetry: nodes may have internal DNS names—within their local institution's network infrastructure—that differ from their external name, perhaps due to NAT; this is particularly problematic in the face of infrastructure failures.
- Non-static addresses and multihoming: both can introduce further complexity into node naming if the node derives its name from its IP address.

The consequences of these problems are twofold: external entities may not be able to identify and access a node through its DNS name, and a node that attempts to determine its own DNS name from its network configuration cannot reliably do so. For example, some PlanetLab nodes had `CNAME` (alias) records that were provided only by certain nameservers—when the primary nameserver failed and the node asked one of these servers for its name (using its IP address) it got back a different result than it would have gotten from the primary nameserver. Similarly, when the reverse mappings for two distinct IP addresses were erroneously associated with the same name we found that two nodes believed they had the same name.

An obvious alternative to using DNS for naming nodes is to use IP addresses, but unfortunately they aren't much better—they do occasionally change, even on 'static' networks e.g., due to local network reconfiguration, and the inherent non-human readable nature of IP addresses always leads to the temptation to convert to DNS names for various purposes, thus introducing DNS-related problems, such as high latency for lookups and reverse lookups, into unrelated code e.g., formatting debug messages.

In PlanetLab we adopted unique numeric IDs as node names, with those IDs being generated by our centralised database whenever a node is installed; a node ID is associated with the MAC address of the primary network adapter, so node IDs do occasionally change, but we have found this scheme to be more reliable than either DNS names or IP addresses for naming nodes.

3.3 Local Clocks are Unreliable and Untrustworthy

A second problem often encountered in distributed systems is maintaining a globally consistent notion of time [4]. Specific requirements in PlanetLab that have

proven problematic include determining whether an externally generated timestamp is in the past or future, and monitoring whether a particular subsystem is live or dead. Of course, as a testbed for network experiments it is also imperative that PlanetLab provide a reliable clock for measurement purposes.

The root cause of time-related problems is the fact that local clocks are sometimes grossly inaccurate: we commonly observed timer interrupt frequency errors of 2–3% on some nodes, most likely due to bad hardware or interrupts being lost due to kernel bugs, and in the most extreme cases we observed nodes 'losing' about 10% of their timer interrupts over a 1-hour period.

NTP [5] really helps but some sites block NTP ports, and occasionally on heavily loaded systems, which almost all PlanetLab nodes are (see Section 3.6), NTP doesn't run frequently enough to compensate for massive drift. While ideally kernel bugs would be identified and fixed we have found that short-term solutions, such as adjusting the timer period to compensate for lost interrupts, can make a significant difference. Finally, some applications e.g., local measurements, don't actually require a globally correct clock, just a timer of known period for which a facility like the IA-32 timestamp counter is perfectly adequate.

The magnitude of clock errors that are considered reasonable places limits on the granularity at which timestamps are useful. For example, once we realised that the NTP service on our nodes was frequently making adjustments of tens of seconds every 15–20 minutes it becomes impractical to allocate resources with expiration times in the minute range—an NTP adjustment can advance the local clock so far that a resource just allocated becomes expired immediately. Furthermore, although NTP limits the size of its adjustments (usually to ± 1000 seconds), an administrator who sees that a node's clock has deviated far from the correct value will often manually reset the clock, sometimes by as much as several days—applications must therefore be designed and implemented to detect gross changes in local time and respond gracefully.

Similarly, external monitoring of nodes e.g., to detect when an event was last recorded, is unreliable if the timestamp recorded with the event is not known to be consistent with the monitor's notion of time. The solution depends upon the particular type of monitoring: for live services we have found that having a simple 'ping' facility in the application that can be exercised by the monitor is preferable, while the accuracy of timestamps in, say, log files can be increased somewhat by considering the current difference between the monitor's clock and the monitored node's local clock, if that difference is assumed to be representative, or used to calculate the appropriate value.

3.4 Inconsistent Node Configuration is the Norm

It's hard to maintain a consistent node configuration across a distributed system like PlanetLab. Typically a significant fraction of nodes will not have all the latest versions of software packages and the most up-to-date configuration information, usually because of network outages that prevent the auto-configuration service from downloading new files. In addition to leaving nodes exposed to security vulnerabilities if recent updates have not been applied, application designers must explicitly handle inconsistencies between multiple versions of data files and software packages; we focus only on the latter problem.

The biggest effect of this global inconsistency is that system administrators cannot make drastic changes to data distributed to the nodes without taking into account the ability of old versions of node software to handle the new data (and corresponding metadata e.g., file formats). This problem is exacerbated by the fact that software and configuration updates may not be well-ordered: a change to a configuration file may be applied to a node even though a prior change to the corresponding software package has not been applied.

For example: in PlanetLab we maintain a centralised database of user *slices*—high-level virtual machines—that are distributed across PlanetLab nodes. The contents of this database are exported to nodes via an XML file retrieved using HTTP (over SSL). When changes are made to the database and/or the subset of that database exported in that XML file we have to consider how our changes will affect nodes running out-of-date versions of the Node Manager software—if the format of the slice description changes will NM fail to recognise that a slice should be instantiated on the local node, and furthermore delete it from that node if already instantiated?

Applications such as NM can be made more robust against unexpected data format changes to a certain degree, and it is often also possible to incorporate failsafe behaviour, so that, say, the sudden disappearance of all slices from the XML file, is recognised as most likely being due to a major format change. The most obvious way to prevent unexpected behaviour due to signficant changes is to associate version numbers with file formats, protocols, APIs, etc., so that large changes can be easily detected. But it is generally not possible to completely isolate data publishers—those entities that push data out to the nodes—from the need to consider the effects of data format changes upon application software.

3.5 There's No Such Thing as "One-in-a-Million"

In a distributed system with hundreds of nodes running 24/7, even the most improbable events start to occur on a not-too-infrequent basis. It's no longer acceptable to ignore corner cases that probably never occur—those cor-

ner cases will occur and will break your application.

For example: the Linux *logrotate* utility rotates log files periodically. It can be configured to send a `SIGHUP` signal to the daemon writing a log file when the log file is rotated, so that the daemon can close its file handle and reopen it on the new file. If the daemon happens to be executing a system call when the signal is received that signal call will be terminated with an `EINTR` error which must be handled correctly by the daemon. If logs are rotated only once a day or even once a week, and the daemon only spends 1% of its time executing system calls, then it becomes very tempting to ignore the possibility that the signal will be received at the most inopportune moment; unfortunately, as we discovered, this possibility will actually happen with a non-negligible frequency.

A similar problem arises with filesystem corruption due to nodes being rebooted without being shutdown properly, an event that also occurs much more frequently than one's intuition leads one to believe. Whereas a user running Linux on their desktop only very infrequently finds their system rebooted involuntarily e.g., due to power outages, in a distributed system located at hundreds of sites it is not uncommon for there to be one or more such outages every week. Consequently we find that instances of filesystem corruption are not uncommon, particularly since our nodes are frequently very heavily loaded and so the probability of there being inconsistent metadata when such an outage occurs is high.

Hence application developers must not cut corners when handling error conditions, and must be careful that their own personal experiences of using one or two computers at one or two locations do not lead to incorrect assumptions of whether unusual events will occur.

3.6 No PlanetLab Node is Under-Utilised

We observe that even with an increasing number of PlanetLab nodes, each one is still more or less fully utilised at all times. CPU utilisation is typically 100%, with twenty or more independent slices concurrently active within even short time periods (on the order of a few minutes). Hence it is not uncommon to find several hundreds of processes in existence at any instant in time, and load averages i.e., number of runnable processes, in the 5–10 range are normal. This is yet another example where user experience running an application on a single-user workstation doesn't provide good intuition as to how that application will behave when deployed on PlanetLab.

The most obvious effect of this over-utilisation is that applications typically run much slower on PlanetLab, since they only receive a fraction of the CPU time available. While this is often not in itself a problem, it can have unforeseen implications for functions that make implicit assumptions about relative timing of operations. For example, the PLC agent component of Node Manager uses a pair of RPC operations to create new slices: in an unloaded system the first operation—acquiring a resource handle—completes very quickly, so the resource corresponding to the new handle is likely to be available for use by the second RPC operation. When the system is heavily loaded, the latency of the first operation can increase up to several minutes or more, so the state of the system when the second RPC is invoked may have changed significantly and the resource referred to by the handle may no longer be available.

The inherent global nature of such resource overconsumption means that a system-wide solution is often most effective; in PlanetLab we implemented a proportional share CPU scheduler to guarantee that each slice gets an equal (or weighted) share of the CPU within every time period (typically a few seconds). However, it is still prudent for applications to anticipate the effects of heavy load: fortunately many of the enhancements that are appropriate are similar to those concerned with inaccuracies in local clocks e.g., not allocating resources with very short timeouts. Another measure adopted by Node Manager to handle heavy load is to detect when an operation takes too long (obviously defined in an application-specific manner) and build a degree of leniency into the system e.g., extend resource expiration times in such cases.

3.7 Limited System Transparency Hampers Debugging

A further difference between developing an application on a single-user workstation and deploying it on a distributed system like PlanetLab is that one often does not have complete access to system nodes in the latter environment. While some distributed systems only provided users with restricted accounts, in other cases the lack of transparency is due to virtualisation that prevents the illusion of full access while hiding other system activity from the user.

For example, in PlanetLab we allocate user slices as environments within which services and experiments are run, each slice being an isolated VM that gives the appearance, albeit only if one doesn't look *too* closely, of each user having unrestricted access to a Linux system; each user has a restricted root account within their slice that can be used to run standard system configuration tools such as RedHat's `rpm` or Debian's `dpkg`.

Unfortunately the shared nature of PlanetLab nodes combined with the limited view of the system available to each user can make it more challenging for individual users to debug their applications. Similarly to the previous point, this problem is best addressed by system-wide measures, such as providing new tools that can be used by slice users to obtain the required debugging and status information. One example is the PlanetLab *SliceStat* ser-

vice that exports aggregate process information for each slice i.e., the total amount of memory, CPU time and network bandwidth consumed by all processes within a slice.

From the individual user's perspective, this lack of visibility into the distributed system emphasises the importance of debugging the application locally as thoroughly as possible before deploying to PlanetLab, ideally under simulated conditions that closely resemble those of the distributed environment e.g., heavy CPU load. Reporting as much information as is available to the application rather than relying upon traditional system monitoring tools—dumping received packets in a readable manner rather than using an auxiliary `tcpdump`, say—also pays dividends.

4 Conclusions

Application developers who are designing and implementing new distributed applications, or porting an existing application, have a number of challenges inherent to the distributed environment to address. Our experiences implementing the Node Manager component of the PlanetLab infrastructure led us to develop a list of the seven most important of these challenges. From the solutions adopted to address these challenges a smaller set of general guidelines emerged:

- Many assumptions made in non-distributed applications are not valid in large-scale and/or heterogeneous environments.
- Distributed applications must gracefully handle a broad variety of corner-case failure modes that are often ignored in the non-distributed environment.
- Resource management in the distributed environment is significantly different from the non-distributed case.
- Even local operations can behave radically differently in a system that is heavily over-utilised.

By following these guidelines in the implementation of Node Manager we have successfully increased the robustness of the distributed application and helped make PlanetLab a more reliable environment for deploying new network services.

References

[1] ANDERSEN, D., BALAKRISHNAN, H., KAASHOEK, F., AND MORRIS, R. Resilient Overlay Networks. In *Proc. 18th SOSP* (Banff, Alberta, Canada, Oct 2001), pp. 131–145.

[2] CHUN, B., ET AL. PlanetLab: An Overlay Testbed for Broad-Coverage Services. *ACM SIGCOMM Computer Communication Review 33*, 3 (Jul 2003).

[3] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The Google file system. In *Proc. 19th SOSP* (Lake George, NY, Oct 2003).

[4] LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM 21*, 7 (Jul 1978), 558–565.

[5] MILLS, D. L. Internet time synchronization: The Network Time Protocol. *Internet Req. for Cmts.* (Oct. 1989).

[6] PETERSON, L., ANDERSON, T., CULLER, D., AND ROSCOE, T. A Blueprint for Introducing Disruptive Technology into the Internet. In *Proc. HotNets–I* (Princeton, NJ, Oct 2002).

[7] WATSON, R. W. Identifiers (Naming) in Distributed Systems. In *Distributed Systems—Architecture and Implementation, An Advanced Course*. Springer-Verlag, 1981, pp. 191–210.

[8] WHITE, B., LEPREAU, J., STOLLER, L., RICCI, R., GURUPRASAD, S., NEWBOLD, M., HIBLER, M., BARB, C., AND JOGLEKAR, A. An Integrated Experimental Environment for Distributed Systems and Networks. In *Proc. 5th OSDI* (Boston, MA, Dec 2002), pp. 255–270.