Autopilot: Automatic Data Center Management

Michael Isard
Microsoft Research Silicon Valley
misard@microsoft.com

ABSTRACT
Microsoft is rapidly increasing the number of large-scale web services that it operates. Services such as Windows Live Search and Windows Live Mail operate from data centers that contain tens or hundreds of thousands of computers, and it is essential that these data centers function reliably with minimal human intervention. This paper describes the first version of *Autopilot*, the automatic data center management infrastructure developed within Microsoft over the last few years. Autopilot is responsible for automating software provisioning and deployment; system monitoring; and carrying out repair actions to deal with faulty software and hardware. A key assumption underlying Autopilot is that the services built on it must be designed to be manageable. We also therefore outline the best practices adopted by applications that run on Autopilot.

Categories and Subject Descriptors
C.2.4 [**Computer-communication networks**]: Distributed systens – *distributed applications*, *network operating systems*

General Terms
Management, Design

Keywords
Automatic management, Cluster computing

1. INTRODUCTION
Microsoft is rapidly expanding the scope of its web-scale online services. Windows Live Search was re-launched using an internal back-end in January 2005, and Windows Live Mail (formerly Hotmail) has seen a large growth in storage capacity over the last few years. Several more web-scale services are currently in development, and the total number of server computers managed by Microsoft has increased very quickly over the last few years, and will continue to grow.

The sudden growth of Microsoft's data center capacity at the same time that several new service back-ends are being developed has given us an opportunity to design a new in-house infrastructure for automatic data center management. This infrastructure is known as *Autopilot*. Its design was primarily motivated by the need to keep the total cost of a data center, including operational and capital expenses, as low as possible. This is partly achieved by using more intelligent software to replace much of the repetitive work previously handled by operations staff. We aim to maintain as few people as possible on 24-hour call: our most efficient services support many thousands of computers per member of operations staff, and run on an 8x5 rather than 24x7 support schedule.

Increased reliability is an equally important benefit of automation. Many data center failures are caused by human error, often resulting from an attempt to fix an earlier problem. As more failure management is moved to automated scripts, there is less variability in the response to faults, and thus we can hope to make the entire system more reliable and maintainable.

The first version of Autopilot described here concentrates on the basic services needed to keep a data center operational: provisioning and deployment; monitoring; and the hardware lifecycle including repair and replacement. Autopilot supplies mechanisms to automate all of these services, however policy — for example, determining which computers should run which software, or precisely defining and detecting failures that need to be repaired — is mostly left to individual applications. Converting legacy applications to work with new automatic management software is a challenging problem. Autopilot has the luxury of starting from a fresh application base, supporting mainly systems that were built to conform to Autopilot's design principles.

Most of the technology used in Autopilot components is similar to designs that have appeared in previously reported work. Our overall approach to fault tolerance follows the Recovery Oriented Computing model outlined in [3], and we adopt the crash-only software methodology proposed in [4]. Our software deployment strategy fits the framework advanced in [1]. There has been much recent interest in "autonomic" computing, and a survey of commercial work in this area is given in [5], but the goals of that community are more ambitious than those of Autopilot. Autonomic computing looks forward to the day when most configuration *policy* will be controlled automatically. As noted above, this paper describes mostly mechanism to support manually-determined policies. The factors that underlie our desire to implement fault-tolerance in software on top of a large number of unreliable commodity computers are similar to those described in [2]. Many of the design practices we follow are standard software-engineering methodology. Nevertheless, it is still a challenge to combine all of these ideas into a fully automatic system that can manage tens of thousands of computers 24 hours a day for years at a time without any planned downtime for the system as a whole.

This paper gives an overview of Autopilot's structure, but it does not attempt to fill in the details: it would be impractical to try to supply enough information here to allow even a skilled practitioner to re-implement the whole system. Instead we have tried to abstract and explain some of the high-level design principles we adopt that let us write and maintain complex software deployed in large-scale modern data centers. Section 2 outlines our design philosophy. Section 3 describes the typical hardware configuration of our data centers. Section 4 gives an overview of Autopilot's component structure, and Sections 5 to 7 examine each component in more detail. Section 8 provides a brief case study showing how one application interacts with Autopilot, and we discuss some lessons we learned along the way in Section 9.

The named author participated in the design of several Autopilot components, however this paper is primarily a report on the work of others. The original conception, the vast bulk of the design, and all the implementation of Autopilot were undertaken by product groups at Microsoft, led by the Windows Live Search core team.

2. DESIGN PRINCIPLES

Traditionally, reliable systems have been built on top of fault-tolerant hardware. The economics of the contemporary computing industry dictate, however, that the cheapest way to build a very large computing infrastructure is to amass a huge collection of commodity computers. In exchange for lower capital expenditure compared with the traditional approach, this results in hardware that is much more prone to failures. This as an opportunity to move more fault-tolerance into software, but we must employ consistent design principles in order to be confident about the reliability of the applications we deploy.

The two most important principles underlying the Autopilot design are fault tolerance and simplicity:

- Since any component can fail at any time, the system must be reliable enough to continue automatically with some proportion of its computers powered down or misbehaving. All vital state must be replicated, and any necessary fail-over must be completely automatic. We aim to minimize critical dependencies between components so that a temporary fault in one service does not become a single point of failure and disable an entire cluster.

  The basic failure model we have assumed is non-Byzantine. This is a consequence of the controlled environment within our data centers. Data corruption problems can generally be managed using checksums, so Byzantine faults tend to arise when some replicas violate a protocol contract. Although this type of failure does occur, it is more likely to be caused by bugs than by the malicious hijacking of a small number of processes. Consequently, problems are typically not confined to a minority of replicas, and so Byzantine fault-tolerant algorithms are of limited usefulness. We briefly revisit this issue in Section 9.

- We believe that simplicity is as important as fault-tolerance when building a large-scale reliable, maintainable system. Often this means applying conservative design principles: in many cases we rejected a complex solution that was more efficient, or more elegant in some way, in favor of a simpler design that was good enough. This requires constant discipline to avoid unnecessary optimization, and unnecessary generality. "Simplicity" must always be considered in the context of the entire system, since a solution that looks simpler to a component developer may cause nightmares for integration, testing, or operational staff.

  Simplicity is also manifested in more basic ways. Where components are configurable, the parameters are stored in human-readable plain text files that are under the management of the same version control system as source code and documentation. (Autopilot components never use the Windows Registry.) If a change in configuration is made, this is done by a deployment procedure (see Section 5.2) with an audit trail. We discourage the use of any interactive control channel to a process that would allow configuration changes to be made without generating an audit trail.

Where correctness is at stake we attempt not to cut corners even when it introduces extra complexity. Of course every design is only as "correct" as the assumptions on which it is based, including the assumption that there are no bugs in the code. No algorithm can provide hard guarantees of correctness in a practical system built on physical hardware, so it is impossible to do more than provide "best effort" service. We can however distinguish between designs that make their assumptions explicit and those that make implicit assumptions, for example that failures will not happen in "pathological" combinations. In a large system one must expect all combinations of failures, so we aim to use designs whose assumptions we understand, but that are simple enough that we can hope to find all crucial bugs through careful review and testing. At the same time we accept that no implementation is foolproof, and the best we can achieve is to optimize for a tolerable level of risk.

Our fault tolerance strategy requires that components must be designed so any process can be killed unexpectedly without destabilizing the system. Most of our components therefore treat forced termination as the *only* exit mechanism and can consequently omit clean shutdown code. Because processes must be able to tolerate crashes, we are able to use assert statements very liberally and, along with the resulting debugging benefits, this can also help to simplify a design since there is no need to try to recover from a damaged invariant. This is a special case of a general principle of avoiding seldom-used failure paths in our programs.

As explained in the Introduction, it was not a design requirement that we provide all of the benefits of Autopilot to legacy code. Legacy applications often assume reliable hardware, but by designing new applications with automation in mind we can move much of this reliance into software and thus reduce hardware complexity and cost. Partly, this means advocating to application designers the same principles of fault tolerance and simplicity that are adopted in the Autopilot components. Applications must expect their processes to be killed without warning, and where possible, customer-facing services should continue, perhaps with degraded operation, in the face of even large numbers (e.g. 50%) of failed computers. Applications must use Autopilot interfaces for reporting errors in order to benefit from automatic monitoring and failure management. Applications must also be easy for Autopilot to install and configure: in practice this means that an application configuration must be entirely specified by files in the local file system of the computer where the application is running.

3. HARDWARE CONFIGURATION

In common with other contemporary data center operators, we buy and install computers in quantities of at least a rack. Each computer conforms to one of a small set of standard specifications including, for example, an "application" configuration with several processor cores and a few direct-attached hard drives, and a "storage" configuration with more disks per processor. A typical "application" rack might contain 20 identical multi-core computers, each with 4 direct-attached hard drives. Also in the rack is a simple switch allowing the computers to communicate locally with other computers in the rack, and via a switch hierarchy with the rest of the data center. Finally each computer has a management interface, either built in to the server design or accessed via a rack-mounted serial concentrator. This ensures that, at a minimum, it is possible for a remote software component to

power each computer on and off and install new Operating System images.

The set of computers managed by a single instance of Autopilot is called a *cluster*. At this point, the largest deployed Autopilot clusters contain up to tens of thousands of computers, though many are much smaller. There may be more than one cluster in a data center, but we aim as far as possible to eliminate inter-dependencies so that a failure in one Autopilot cluster is unlikely to affect other services.

4. AUTOPILOT SYSTEM OVERVIEW

Autopilot is divided into several components, illustrated in Figure 1. This simplifies the overall design, and makes it easier to modify parts of the Autopilot independent of the rest. However, because the system is distributed, this organization has the potential to introduce inconsistencies between the components.

Where information needs to be shared between components in a distributed system, there is often a choice between designs that allow weak consistency and those that require strong consistency. Weak consistency can improve availability, since one component may be able to operate for a while from cached data when another is unavailable. However, strong consistency often allows simpler designs. In the absence of strong consistency, faults such as a transient partitioning of the network may allow components to perform conflicting actions that must be resolved when the partitioning is repaired. This leads to a need for conflict detection and recovery logic, introducing extra complexity and adding code paths that are not used except in failure cases.

The shared state held in Autopilot is deliberately kept fairly small, and the system is designed to tolerate latencies of tens of seconds or more for most actions. These choices allow us to make a simple tradeoff between weak and strong consistency, as follows. All of the information about the "ground truth" state that the system should be in, along with the logic to update this ground truth, is held in a single strongly-consistent state machine called the *Device Manager* that is typically distributed over 5–10 computers. It is built on a well tested replicated state machine library with a strict abstraction boundary so that bugs in the replication mechanism are decoupled from bugs in the state machine implementation. The library is lightweight (around 5,000 lines of C++ in total) and uses the Paxos algorithm [6] to achieve consensus between replicas, batching updates in order to get a reasonable balance between latency and throughput.

The Device Manager stores the state that the system should be in, however it does not itself take any actions to keep the cluster synchronized with this ground truth. Instead Autopilot includes a number of "satellite" services. These satellites lazily perform actions to bring themselves or their clients up to date when they discover that the Device Manager state requires it. Similarly, if a satellite discovers a fault or inconsistency in the cluster, it will not attempt to rectify it but instead will report the problem to the Device Manager. This allows Autopilot to integrate information across the cluster and apply system-wide throttling or consistency checks before deciding to take action. At that point, the Device Manager updates its state, which will eventually result in a satellite repair service noticing the update and taking appropriate action.

The satellite services are themselves replicated, and may contain private state. The Autopilot design partitions system state between satellites and the Device Manager in such a way that the satellites can keep their state weakly consistent without compromising correctness.

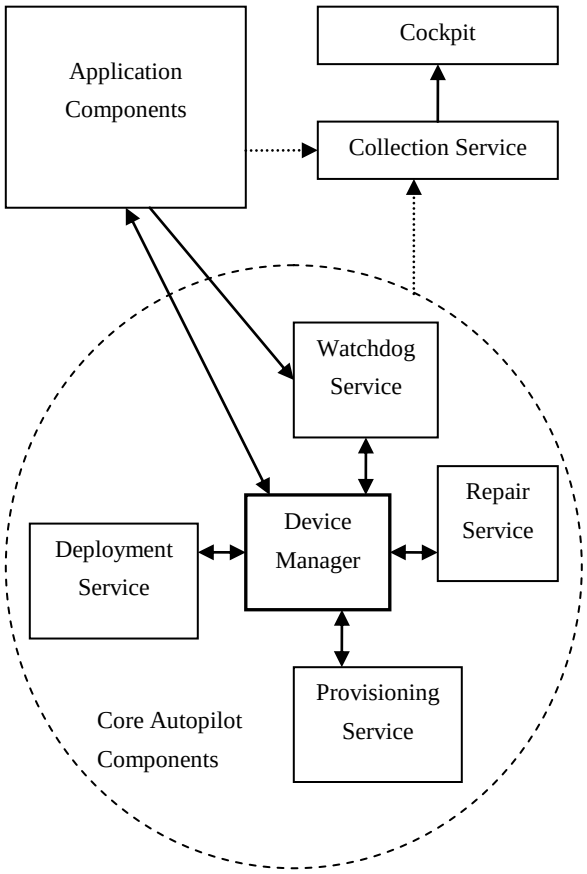

Figure 1: A schematic of the Autopilot system and applications. Arrows show the flow of communication. The Device Manager (Section 4) is the central system-wide authority for configuration and coordination. The Provisioning Service and Deployment Service (Section 5) ensure that each computer is running the correct operating system image and set of application processes. The Watchdog Service and Repair Service (Section 6) cooperate with the application and the Device Manager to detect and recover from software and hardware failures. The Collection Service and Cockpit (Section 7) passively gather information about the running components and make it available in real-time for monitoring the health of the service, as well as recording statistics for off-line analysis. (These monitoring components are "Autopiloted" like any other application, and therefore communicate with the Device Manager and Watchdog Service which provide fault recovery, deployment assistance, etc., but this communication is not shown in the figure for simplicity.)

Satellite services receive information from the Device Manager using a "pull" model. They regularly send lightweight messages to the Device Manager that report their current status, and in response are sent the relevant parts of the current Device Manager ground truth. The use of regular "heartbeat" messages makes the

design very robust to transient failures, since individual messages can be lost without affecting eventual correctness. Sometimes a state transition in the Device Manager will cause it to "kick" remote services to request that they send a "pull" query immediately. This is simply an optimization that lets us ensure that most satellite computers quickly learn about any required actions, but any computers that do not receive the kick will still learn of the state change through a later heartbeat. An alternative "push" model would require the Device Manager to keep state for every message recording which clients had so far received it: we decided that the extra network traffic and latency incurred by the "pull" design was an acceptable tradeoff in exchange for a simpler Device Manager.

5. LOW-LEVEL SERVICES

A small number of operating system images are in use at any time in a cluster. We use only stable commercial releases of Windows Server operating systems. Each image also contains some basic Autopilot configuration files, and some Autopilot-specific Windows services, pre-installed and enabled on boot. The configuration files contain, for example, the DNS names of computers running core Autopilot components. The Windows services are able to communicate with centralized Autopilot components to ensure that the correct application processes are installed and running on the computer. Network configuration and name services are currently managed independently of Autopilot using a standard replicated installation of Active Directory [7].

Every computer runs a local service, supplied by Autopilot, that ensures the correct files are present on its local disk. This *filesync* service is used extensively by Autopilot and applications to transfer data between computers. The service acts both as a client requesting files from remote machines, and as a server handling such client requests. By using a dedicated service rather than relying on an operating system component (such as the standard Windows remote file access features) we gain better control over logging of file transfers, and the ability to throttle transfers so a computer or switch does not become unexpectedly overloaded.

A second local service on every computer, called the *application manager*, makes sure that the correct processes are running. Each application process is distributed as a standalone directory containing all binaries, shared libraries and configuration files necessary to for the process, along with a standard 'start.bat' script that can be invoked to start it. There is no clean shutdown code so a process is stopped simply by instructing Windows to kill it and its children. The application manager reads a configuration script and ensures that the designated binaries are running. A process can be configured to run continuously (so it is restarted if it exits for any reason) or periodically, e.g. once an hour.

5.1 Operating System Provisioning

The *Provisioning Service* includes basic services such as DHCP and network boot. It also contains a scanner that constantly probes the network looking for new computers that have been plugged in. On finding one, it consults the Device Manager to learn what operating system image it should be running, then uses the computer's management interface to install and boot the image, and run burn-in tests. If these steps succeed, the Provisioning Service informs the Device Manager. The newly provisioned computer independently contacts the Device Manager to learn what application binaries to fetch and run, as described in the following sections. The computer's name is determined by its position in the network hierarchy, which in turn is determined by the rack (and slot in that rack) where it is plugged in. It is up to the operator to ensure that the correct hardware configuration is installed in each rack slot.

The Provisioning Service is configured to use several computers for redundancy. These cooperate to elect a leader that carries out the appropriate actions. The Provisioning Service is stateless, and any necessary information is retrieved from the Device Manager when the leader starts up. Weak consistency of the deployment state during fail-over may cause some actions to be attempted more than once, but this can be tolerated since any resulting problems will eventually be detected and corrected by the normal activity of the repair services.

5.2 Application deployment

The application defines a set of *machine types* that are present in the cluster. Each type corresponds to a particular role that the computer might take on: for example in Windows Live Search these include "web crawler," "front-end web server," etc. The Device Manager database stores the machine type of every computer in the cluster. This mapping from computer to machine type is currently manually and statically configured. Autopilot does not perform migration or load balancing, though of course applications frequently implement their own load balancers to automatically distribute work among the computers of a given machine type.

The only difference between machine types from the perspective of deployment is the set of configuration files and application binaries that should be present on the computer (in particular, a machine type is not necessarily tied to a particular hardware configuration). The application manager is responsible for reading the configuration and making sure the appropriate processes are running. Each such set of files is described using a configuration *manifest*, stored in the Device Manager database. A computer can store multiple manifests at once[1], with one "active," and the application manager will run the processes listed in the active manifest. This allows new application versions to be pre-loaded onto computers before they are scheduled to be enabled; also recent previous versions are kept for a while to allow rapid rollback to a known good state in the event of problems. The list of manifests that should be stored on each computer is recorded in the Device Manager along with the name of the computer's active manifest.

The *Deployment Service* is a set of weakly consistent replicas each of which contains a set of manifest directories containing the files listed in each manifest. These directories are populated by a build system outside the cluster. Each computer in the cluster runs a periodic task that queries the Device Manager database to learn what manifests should be present on its disk. If any manifests are missing, they are fetched from one of the Deployment Service replicas, picked at random. If any manifests are present on the disk but are not on the list returned by the Device Manager, they are deleted. The computer informs the Device Manager when its

[1] We refer with slight abuse of terminology to a computer "storing a manifest," meaning that the computer stores the files listed in that manifest.

manifests are up to date, so the Device Manager contains both a central record of what versions should be on each computer, and a weakly consistent view of the versions that are actually present.

5.3 Deploying new code

In the steady state, each machine type in the cluster is associated with a single active manifest, so every computer with the same machine type is running the same set of application binaries, with identical configurations. When a new version is ready for deployment, the new manifest is stored to the Deployment Servers and the Device Manager is instructed to roll it out on the specified machine type. It immediately adds this manifest to the storage list for each computer with the designated machine type, and kicks each computer to tell it to fetch it. Computers that do not receive this message for any reason will fetch the manifest later when their scheduled synchronization task runs. When a high enough proportion of computers have fetched the new manifest, the Device Manager is ready to start instructing computers to make the new version active.

Autopilot defines the concept of a "scale unit" which is a set of up to around 500 computers. In general the computers in a scale unit encompass multiple machine types. The purpose of partitioning the computers this way is to allow staged rollouts of a new code version. Each machine type is configured with a maximum number of scale units that Autopilot is allowed to concurrently modify. If an application component uses 1000 computers, these might be partitioned into ten 100-computer scale units, and the computers with a given machine type might be spread evenly across the scale units. Autopilot could then be configured to roll out at most 3 scale units, or 30% of the computers with that machine type, at a time.

When the Device Manager decides to roll forward the set of computers in a scale unit with a particular machine type, it updates its database to instruct the computers to run the new manifest, then kicks them to synchronize their configuration. The Device Manager monitors the computers and learns when each is successfully running the new version. The mechanism to determine this is described in Section 6. When a high enough proportion has successfully rolled forward, the machine type rollout for that scale unit is declared to be successful, allowing the next scale unit's rollout to proceed. If a timeout is reached before a sufficient proportion of computers has successfully rolled forwards, the rollout is cancelled, and all computers (in all scale units) with that machine type are rolled back to the old version, one scale unit at a time.

Deployments are started with a single operator command and then proceed fully automatically and unmonitored, often overnight if many computers are involved. In the event of a rollback, an operator can determine the cause at a later date and take appropriate action before attempting the rollout again. Even simple configuration parameter changes are handled using the deployment machinery, and therefore a mis-configuration is rejected by automatic rollback exactly as a buggy code version would be. Likewise, new operating system images are deployed in the same way.

6. AUTOMATIC REPAIR SERVICES

Autopilot supports a simple model for fault detection and recovery. It was designed to be as minimal as possible while still keeping a cluster operational. The unit of failure is defined to be a computer or network switch rather than a process, and the only available remedies are *Reboot*, *ReImage*, and *Replace*, described in more detail in Section 6.2 below. This ensures that Autopilot does not need to contain logic to attribute blame to a particular process or component, or consider application-specific recovery actions.

6.1 Watchdogs

Faults are detected using a set of watchdogs. A watchdog probes one or more computers to test some attribute, and then reports to the Device Manager. The watchdog reports *OK*, *Warning*, or *Error* for the attribute on each computer, along with an arbitrary descriptive reason string for the latter two. The set of watchdogs is extensible; the definition of a watchdog is simply any piece of code that understands how to contact the Device Manager using the watchdog protocol. This is a simple plain-text protocol so it is easy to write a watchdog in a scripting language.

The Device Manager can compute a transient error predicate for any computer using the watchdog attributes: if any watchdog reports *Error*, the computer is in error; if all watchdogs report either *OK* or *Warning*, the computer is not in error. This predicate is used to drive the state machine outlined in Section 6.2. The *Warning* status is used to report unexpected but non-fatal conditions. The audit history of warnings can be useful, for example, during the postmortem analysis of an unexpected event, but a warning does not automatically trigger any Autopilot action.

We could have built an alert system that would contact an operator when it detected warnings. However, a fundamental goal of Autopilot is to *avoid* burdening operations staff with the task of monitoring and understanding alerts, or taking remedial actions. We therefore don't want to encourage developers to use warnings as a "lifeline" to a human: rather, the system should be designed to react to problems automatically. Applications sometimes do need to generate alerts, but they are typically triggered by information integrated from multiple computers or components (see Section 7).

Some watchdogs are supplied by Autopilot and are run either on every computer locally or on a set of computers called the *Watchdog Service*. These standard watchdogs include periodic checks that every computer is running the right Operating System image and manifest; and queries to the computer's BIOS to detect disk or memory error conditions. Other watchdogs are supplied by the application. There is no need to limit the number of watchdogs, so new watchdogs are often added to address specific scenarios. For example, at one point it was discovered that some computer configurations would spontaneously lose track of half of their DRAM and consequently start paging until the symptom was cured with a reboot. This was addressed with a custom watchdog to periodically probe for the issue.

The Device Manager error predicate is the conjunction of all the watchdogs for a computer, and once a computer is held to be in error, it may be unavailable for a substantial period. It is therefore important to minimize false positives in watchdogs. On the other hand, when there is a fault that is detectable using a watchdog there may be minutes of latency before the Device Manager discovers it and takes action. Components that require low-latency fault mitigation therefore typically implement custom "soft-failure" detectors and Section 8 explains this in more detail with reference to a specific example component.

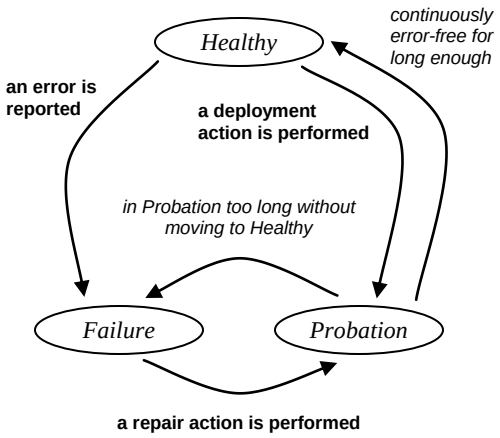

Figure 2: A simplified diagram of the failure/recovery state machine. The Device Manager records its estimate of the state of each computer in the cluster as *Healthy*, *Failure* or *Probation*. Transitions described in bold text occur as a side effect of other Device Manager actions. Transitions described in italics occur when a timer expires. The state machine is described in Section 6.2.

6.2 The Failure/Recovery State Machine

The lifecycle of a computer's repair state is shown in Figure 2. A computer that is functioning normally is marked by the Device Manager as being in the *Healthy* state. If any watchdog reports an error for that computer it is placed in the *Failure* state and assigned an appropriate "recovery action" from the set *DoNothing*, *Reboot*, *ReImage*, *Replace*. The choice of recovery action depends on the recent repair history of the computer and the error that is reported. A computer that has not had any errors for multiple days or weeks and experiences a single application fault might be assigned *DoNothing* in the hope that the fault was transient and isolated. On the other hand a fatal disk error reported by the BIOS will immediately cause a computer to be marked for replacement. Repeated non-fatal errors on the same computer in a short time window will cause the Device Manager to escalate through reboots to re-imaging which involves reformatting a computer's disks and reinstalling the operating system. Eventually a computer will be marked for replacement. Technicians sweep the data center every week or two to remove dead computers and replace them with spares that are then automatically discovered and provisioned as described in Section 5.1. The first version of Autopilot contains simple heuristics to determine the order and timing of repair actions, and carries them out independently without notifying the application. Some applications benefit from advance warning of repairs, and this is discussed in Section 9.

The *Repair Service* periodically asks the Device Manager for the list of computers in the *Failure* state and uses their management consoles to perform the required repair action, after which the Device Manager is told to move the computer to the *Probation* state. In this state a computer is expected to start off with some watchdogs reporting errors. If the repair action was successful, these errors should quickly disappear. If a computer remains continuously error-free in the *Probation* state for long enough, it is transitioned back to *Healthy*. If the computer remains in *Probation* for too long it will be moved back to *Failure* triggering another repair action.

When the Device Manager takes some action such as code deployment (Section 5.3) that is likely to generate application-specific watchdog errors, the set of affected machines is moved from *Healthy* to *Probation* before the rollout is started. The rollout is deemed to be successful on a computer if it undergoes the normal transition back to *Healthy*. If a computer stays in *Probation* for too long then the rollout has failed. This re-uses the existing failure-detection machinery for rollout monitoring, while ensuring that computers do not get assigned a "black mark" in their repair history due to watchdog errors resulting from planned actions.

By centralizing all repair action decisions in the Device Manager state machine Autopilot is able to throttle the number of machines under repair at any time, and therefore protect against, for example, a faulty watchdog causing all computers in a cluster to be simultaneously rebooted.

7. MONITORING SERVICES

Autopilot components, and applications built to run on Autopilot, record performance counters and logs in a standard location on every computer. Performance counters are used to record the instantaneous state of components, for example a time-weighted average of the number of requests per second being processed by a particular server. Performance counter histories are useful for off-line trend analysis, but real-time values are also invaluable to give operators a current view of the state of the system and help in the diagnosis of any unexpected issues. Logs are mostly used to record individual component actions that can be correlated later in off-line processing.

The *Collection Service* forms a distributed collection and aggregation tree for performance counters and logs. It can generate a centralized view of the current state of the cluster's performance counters with a latency of a few seconds. Individual counters can be aggregated, for example across an entire machine type, in order to keep the volume of low-latency data manageable. The Collection Service also lazily collects detailed performance counters and logs and writes them to a large-scale distributed, replicated file store where they are available for off-line data-mining.

Real-time performance-counter information is kept in a SQL database so that sophisticated statistics can be computed for visualization and diagnosis simply by issuing the appropriate relational queries. *Cockpit* is a visualization tool that lets operators monitor one or more Autopilot clusters using graphs and reports generated from the performance counter databases. It is easy to store default views, or construct custom queries to drill down into a particular issue. Cockpit also serves as a gateway allowing operators to fetch arbitrary log files from individual computers. Together with the ability to monitor computers' performance counters, this provides an audited access mechanism that eliminates most requirements for direct access to data center computers. There is an automated *Alert Service* that sends emails or pages to support staff based on application-defined relational queries against the Cockpit database. These queries can capture system-wide properties by aggregating data from many computers and sub-components.

8. CASE STUDY: INDEX SERVING

Autopilot provides a set of basic services that are sufficient to keep the underlying infrastructure of a cluster in good health. We chose to keep Autopilot simple by using lazy repair actions and a minimalist failure/recovery model. Some application components are able to use the Autopilot failure model directly, however they must be able to tolerate latencies of seconds or minutes between the time that a computer fails and the time that the failure is listed in the Device Manager database.

Applications that need high availability for low-latency customer-facing services may therefore need to layer custom fault tolerance on top of the basic Autopilot components. This section explains some of the ways in which the clusters that return query results to users as part of the Windows Live Search application interact with Autopilot and retain high availability.

Although the Device Manager is extremely reliable, the search application developers chose to keep a private weakly consistent snapshot of crucial information so that query results continue to flow for as long as possible if Autopilot fails. A web-index serving cluster includes machine types such as front-end web servers and back-end partitions of the web index. The ground truth listing the computers in each of these machine types is stored in the Device Manager, however the index serving components store a weakly consistent cache of this information (including the IP addresses of each peer) on every computer, so that they can start up and contact each other autonomously, for example after a data center power failure, even if the Device Manager and Active Directory are disabled. Likewise each computer caches a local copy of the list of computers that the Device Manager considers to be failed to avoid wasting time trying to contact those computers; however the application maintains its own internal "soft-failure" list to route around computers that are misbehaving but may still be considered healthy by Autopilot.

There is a constant stream of queries to the Windows Live Search service, and load balancers in the web servers ensure that every computer in the back end is serving some fraction of that query load at any given time. These load balancers can therefore detect very quickly if a computer has stopped responding or is slower than its peers. There is substantial redundancy in the web index, so if a load balancer is suspicious about a computer it can send duplicate queries to redundant partitions: thus the user continues to get responses with low latency, but the suspicious computer continues to receive traffic allowing the load balancers to learn if a transient fault goes away without the need for remedial action.

Each web server operates its own load balancer so there are multiple independent estimators of computer failures. The index serving pipeline operates a monitoring service that integrates information from all of the web servers. If there is consensus agreement that a computer is behaving poorly, the monitor acts as a watchdog to tell Autopilot about the error. Due to the amount of information available to it, this monitor is able to report failures with very few false positives.

This design for Windows Live Search has been successful in allowing the application to ignore all details of deployment, repair actions, etc. At the same time the application retains fine control over the definition of real-time computer failure conditions, and operates reliably in the face of temporary outages of even core Autopilot components.

9. LESSONS LEARNED FROM VERSION 1

Autopilot was originally deployed in concert with the Windows Live Search backend. The design in this paper represents a snapshot of the design shortly after that deployment, but the system has been under continuous development since then. We first note some of the lessons from the first version that have guided this development:

- There were many low-priority issues with the first version that have now been fixed; for example it was initially somewhat complex to set up and bootstrap a new Autopilot cluster and this is now much simpler. Likewise there is support for more diverse computer and networking configurations.
- Some initial Device Manager features were too minimal for applications other than Search. For example, applications that need to store highly reliable replicated data need more control over the throttling of *ReImage* repair actions. Such applications may need the chance to try to copy surviving data from non-faulty disks before they are reformatted in order to retain a good balance between the number of replicas and the mean time to data loss. There may be other examples where the interaction between components requires a more complex mechanism to specify Device Manager logic. For example, an application might request that a particular scale unit should not be taken out of service for code deployment if the number of failures in another component is above some threshold.
- Autopilot has so far been used to manage a relatively small number of complex, highly-engineered, vertical applications. We anticipate supporting a growing number of more lightweight applications that, at least in their early stages of development, require only a small number of computers and may be prototyped very rapidly. Consequently we have been increasing the support services offered by Autopilot and we may, for example, offer some standard solution for low-latency failure detection so that applications do not all have to implement custom designs like that set out in Section 8.
- We have also extended support for running applications or replicated services that are geographically distributed across data centers, and so may require coordination between, for example, deployment processes. Future versions of Autopilot may take over the management of name services from Active Directory in order to reduce the number of custom deployment and management tasks in an Autopilot cluster, and this will also require coordination between clusters to present a consistent view of the name space.

Overall we have found our assumptions to be sufficiently close to reality that no substantial redesign has so far been necessary.

As mentioned in Section 4, we protect strongly-replicated data using replicated state machines employing the Paxos protocol to achieve consensus. This design assumes that failures that cause data loss are independent, but transient non-destructive failures (such as power outages) may be correlated. Both of these assumptions have held up; in particular, any design that did not allow recovery after all computers are shut down would be insufficient.

We have not yet confronted any major faults inside the data center that have defeated our fault-tolerance mechanisms, but that would have been contained by a Byzantine fault-tolerant algorithm. Avoiding the additional complexity of implementing Byzantine

fault-tolerance therefore seems justified. Our staged rollout procedure does introduce new code (along with its new potential bugs) on a small fraction of a machine type's computers at first. This model fits Byzantine fault-tolerant assumptions very well, and some applications may in future adopt more complex fault tolerance strategies if the risk of state corruption justifies it.

Autopilot contains a number of hand-set thresholds defining the policy for deployments, probation state timeouts, etc. We are currently recording large amounts of data logging the actions that Autopilot takes, along with performance counters capturing the state of client applications when those actions are taken. We are experimenting with machine learning algorithms to analyze these data in order to understand how to improve the policy settings, with the ultimate goal of automating many of the current manual policies.

As with all large-scale deployments, we have encountered failures of every type that we expected, and some we didn't.

- It is vital to keep checksums of all crucial files (for example machine-type manifests) since they will become corrupted. Checksums also allow the detection of hand-edited configurations that were temporarily changed, for example as part of a debugging investigation. Without such automatic detection it is very hard to prevent gradual configuration drift in a large system.
- TCP/IP checksums are weak, and messages will be silently corrupted unless they are protected by additional application-level checksums.
- Networking hardware will malfunction and start flipping large numbers of bits; this both causes a storm of retries, and makes it inevitable that some errors will remain undetected by TCP.
- Computers will spontaneously start running very slowly, but keep making progress, so systems need to tolerate and detect this as well as fail-stop errors.
- Throttling and load shedding are crucial in all aspects of an automated system. Failure detectors must be able to distinguish between the symptoms of failure and overloading, otherwise overloaded computers may be marked as failed and removed from service, amplifying the problem and triggering a cascade of failures that disables the entire application.

Autopilot has been continuously operating its oldest clusters since the pre-release Windows Live Search engine was deployed in 2004. Over that time we have rewritten many components and substituted them in place, without ever bringing down a major production cluster for planned maintenance. Autopilot supports all forthcoming large-scale deployments inside Microsoft, and some legacy services have already been ported to run on Autopilot clusters. Autopilot supports a vastly lower cost of management than legacy Microsoft services, with a very high level of reliability. Up to this point, there has been no major outage of a customer-facing service that can be directly attributed to an Autopilot failure.

10. ACKNOWLEDGMENTS

As mentioned in the Introduction, the design and implementation of Autopilot was led by the Windows Live Search core team. Many members of that team have shared comments and advice to help make this an accurate and representative depiction of the system, however any remaining errors are the responsibility of the author. Autopilot's success has only been possible due to the collaboration and hard work of many product teams spanning developers, testers, program management, and of course data center operational staff at all levels.

I would also like to thank Darren Shakib, Kevin Kaufmann, Martín Abadi, Mike Schroeder, Andrew Birrell and John MacCormick for many helpful comments on improving the content and presentation of the paper.

11. REFERENCES

[1] Ajmani, S., Liskov, B. and Shrira, L. Modular Software Upgrades for Distributed Systems. *20th European Conference on Object-Oriented Programming*, July 2006, 452–476.

[2] Barroso, L.A., Dean, J. and Holzle, U. Web Search for a Planet: The Google Cluster Architecture. *IEEE Micro*, 2003.

[3] Brown, A. and D. A. Patterson. Embracing Failure: A Case for Recovery-Oriented Computing (ROC). *High Performance Transaction Processing Symposium*, October 2001.

[4] Candea, G. and Fox, A. Crash-Only Software. *9th Workshop on Hot Topics in Operating Systems*, May 2003, 67–72.

[5] Gentzsch, W., Iwano, K., Johnston-Watt, D. Minhas, M.A. and Yousif, M. Self-adaptable autonomic computing systems: an industry view. *16th International Workshop on Database and Expert Systems Applications*, August 2005, 201–205.

[6] Lamport, L. The Part-Time Parliament. *ACM Transactions on Computer Systems 16*, 2 (May 1998),133–169.

[7] Microsoft Active Directory for Windows Server 2003. http://www.microsoft.com/windowsserver2003/technologies/directory/activedirectory/default.mspx