Title of the Invention: **System for live-migration and automated recovery of applications in a distributed system**

Abstract Title: **Live migration of applications and file systems in a distributed system**

When migrating applications from one server to another, a protocol handler pauses incoming requests during the migration. Outstanding requests may be terminated after a timeout period. Alternatively, the paused requests may be released if the outstanding requests have not completed after a timeout period. When replicating a file system, snapshots of a source file system are taken at predetermined points in time. These are replicated to the destination file system. After a period when the file system could not be replicated, the most recent version of the file system is identified. The last common snapshot is also identified. Subsequent snapshots are then replicated from the most recent version to the older version. The most recent version may be identified from the average age of the snapshots and the number of changes in the snapshots. Only the snapshots unique to a particular server may be used to determine the most recent version.

---

**Diagram 8 – Distributed protocol handler**

![Diagram of distributed protocol handler](image)

- **Incoming Proxy**
- **Pause request?**
- **Choose backend**
- **Outgoing Proxy**
- **Actual App Server**
- **F not live**
- **F live**

**Server A**

**Server B**

**Protocol Handler**

**Request for file system F**

**Internet or network**
Diagram 1 – filesystem replication

Server A

1

2 changes

2

3 changes

Server B

1

Parent pointers

2

Replication event

Diagram 2 – partition!

Server A

1

2 changes

2

3 changes

3

1 change

Server B

1

2 changes

2

3 changes

3'

4 changes
Diagram 3 – example replication system arrangement
Diagram 4 – Snapshotting replicator

Diagram 5 – Per slave sending replicator

Diagram 6 – Receiving replicator
Diagram 7 – Controller state transitions and permitted periodic loops

- Loading local filesystem state
- Heart new heartbeats
- Heard available data from all servers
  - Runs assertOwnership which emits hosted filesystems messages
- Heard available data and hosted filesystems from all servers
  - Runs checkLoadBalancing
  - Runs checkReplication
  - Runs checkDeadSites
Diagram 8 – Distributed protocol handler

Internet or network

Request for filesystem F

Server A

Incoming Proxy

Pause request? Choose backend

Protocol Handler

Outgoing Proxy

Actual App Server F not live

Server B

Incoming Proxy

Pause request? Choose backend

Protocol Handler

Outgoing Proxy

Actual App Server F live
Diagram 9 – Live-migration state machine transitions & protocol

**Thrower**

- Controller finds slot and initiates live-migration, creates Thrower

  - INIT

    - INIT_SLAVE

      - PREREPLICATION

        - <snapshot observed arriving on both sides>

          - BLOCKING_WAITING

            - <distributed proxies wait for open connections to close, then force them closed, or time out and the live-migration is abandoned>

              - UNMOUNTING

                - REPLICATING

                  - waitforsnapshot

                    - completemoveload

                      - UNBLOCK

                        - Throw complete

**Catcher**

- Controller finds slot and accepts live-migration, creates Catcher (or there is no slot, and the entire live-migration is abandoned)

  - INIT

    - PREREPLICATION

      - prereplicate

        - latestsnapshot

          - continuemoveload

            - BLOCKING

              - shortcut if replication already up-to-date

                - REPLICATING

                  - waitforsnapshot

                    - completemoveload

                      - MOUNTING

                        - UNBLOCK

                          - Catch complete
The following terms are registered trademarks and should be read as such wherever they occur in this document:

VMware

MySQL

Unix
SYSTEM FOR LIVE-MIGRATION AND AUTOMATED RECOVERY OF
APPLICATIONS IN A DISTRIBUTED SYSTEM

Technical Field

The present application relates to a mechanism and apparatus for asynchronously
replicating data pertaining to a set of applications across multiple (virtual or physical)
servers (a cluster) in a high-latency networked system in a partition- and failure-tolerant
manner.

Also, a mechanism and apparatus for both mediating access to the hosted applications
and controlling the aforementioned data replication to enable the applications to be
seamlessly live-migrated between servers in response to changing load and topological
preferences of each application.

Background to the Invention

For an organisation which hosts network-connected applications (including, but not
limited to, companies hosting websites on the Internet), there are two key problems:

1. Components, servers, networks and storage devices can fail, in which case
applications will need to be recovered, perhaps manually, from a secondary data
store (such as a backup at a disaster recovery site). We will refer to this as the
redundancy problem.

2. Load generated by applications can vary significantly over time, for example a
website can experience a spike in traffic, so applications may need to be moved
between servers in order to maintain an acceptable level of utilisation. We will
refer to this as the load-balancing problem.
In the case of the redundancy problem, current solutions include:

- Adding redundancy at the physical hardware level, for example by use of dual-redundant power supplies. Disadvantages to this approach include that it is extremely difficult (i.e. expensive) to completely eliminate single points of failure within a single server, and even if this can be achieved, the system will still have a single point of failure in the operating system or other application software (e.g. the web server or kernel might crash).

- Virtualising the server and replicating every change in memory and system state to a second physical host over a high-speed LAN so that the second host can take over if the first fails, for example with VMware vMotion. Disadvantages to this approach include that virtualisation imposes a performance overhead on applications, that it requires almost the resources of two servers to run (the live one and the replica), and that the replica can only be located geographically locally. Furthermore this approach only works with a shared storage backend, which can be prohibitively expensive. Also this approach cannot be applied between datacentres or on commodity setups without high-speed connectivity between servers.

In the case of the load-balancing problem, current solutions include:

- Manually moving applications between servers when a spike of load occurs. Disadvantages of this approach include that individual servers are vulnerable to spikes in load of any of their hosted applications, which can cause all of the hosted applications on a server to crash, and the need for manual intervention which can delay recovery time significantly.

- Isolating applications which are generating large amounts of load on the system with operation-system level constraints, for example the CloudLinux kernel extensions. Disadvantages of this approach include that if an application experiences a spike in load, that application is effectively taken offline (or made to run very slowly) until it is manually moved to another server.
• The use of load balancer appliances (hardware or software) in conjunction with stateless or semi-stateless application servers and a shared storage backend (SAN), in order to distribute the load of the applications across multiple servers. We will refer to this solution as a "classical cluster". Disadvantages to this approach include that the SAN itself acts as a single point of failure, failures of which may be catastrophic, and that such a cluster cannot operate across geographically diverse regions. Further disadvantages to a classical cluster include needing to implement complex solutions for the "split-brain" problem, where servers become disconnected from each other but not from the shared storage medium, which can cause data corruption, requiring that administrators sets up quorum, fencing or STONITH ("shoot the other node in the head") to physically power off a server if it becomes unresponsive.

Summary of Invention

The present invention solves the redundancy problem in the following way:

• All changes to application state are asynchronously replicated to a configurable number of other servers in the system. Point-in-time snapshots of each application's data are taken within a configurable number of seconds of detection of changes to the application data, and the difference between these snapshots are replicated between the servers. This allows an application to be recovered automatically from a very recent copy of the data in the event that the system detects a failure of a component, server, network device, or even an entire data centre. Since there is no reliance on shared storage, no quorum, fencing or STONITH setup is required.

The present invention solves the load-balancing problem in the following way:

• The load caused by applications is continually measured by the system and used in a distributed decision-making process to initiate seamless live-migrations of
applications between servers. For example, if a server A is hosting applications \{1, 2, 3\} and a server B is hosting applications \{4, 5, 6\}, and both applications 1 and 2 experience a spike in load, while the remainder are quiescent, the system may elect to live-migrate 2 to server B for a balanced configuration of \( A \rightarrow \{1, 3\}, B \rightarrow \{2, 4, 5, 6\} \).

According to a first aspect of the present invention there is provided a system for dynamic migration of applications between servers, the system comprising a plurality of servers for hosting applications, each of the plurality of servers comprising a protocol handler for receiving requests for applications, wherein the protocol handler is configured to pause incoming requests for an application during migration of applications between servers.

The system may further comprise a load balancer for measuring load on one of the plurality of servers caused by one or more applications hosted on that server, the load balancer being configured to initiate migration of one or more applications from the measured server to another server when a predetermined load condition of the measured server is met.

The plurality of servers may each have a controller that maintains a record of the server on which an application is currently hosted, and the protocol handler is configured to inspect the record to determine the server to which an incoming application request is to be directed.

The protocol handler may be configured to pause incoming requests for an application and to terminate current requests for an application after a predetermined time period.
Additionally or alternatively, the protocol handler may be configured to pause incoming requests for an application for a predetermined time period and to release the paused requests if current requests for an application have not completed in the predetermined time period.

According to a second aspect of the invention there is provided a method for replicating a filesystem between a first server and a second server prior to and following a partition between the first server and the second server, the method comprising: at the first server, taking snapshots of a current state of the filesystem at predetermined points in time following modification of the filesystem, each snapshot recording differences between the current state of the filesystem on the server and the state of the filesystem on the server at the time point of a previous snapshot; continually replicating the snapshots taken on the first server to the second server as soon as they are taken; upon detection of a partition, both the first and the second server becoming masters for the filesystem and accepting new modifications to the filesystems; after recovery of the partition, performing an update process to update the filesystem, the update process comprising: identifying which of the first server and the second server contains the most current version of the filesystem; nominating the server so identified as the master server and the other server as the slave server; identifying a snapshot that is common to both the master server and the slave server; and replicating subsequent snapshots from the master server to the slave server.

Identifying which of the first server and the second server contains the most current version of the filesystem may comprise calculating a centre of mass metric for the version of the filesystem on each of the servers, the centre of mass metric representing the average age of the snapshots of the filesystem on each server and the number of changes to the filesystem represented by the snapshots on each server.

Identifying which of the first server and the second server contains the most current version of the filesystem may further comprise identifying a set of snapshots of the
filesystem that for each server, each set of snapshots containing snapshots only present on that server, and calculating the centre of mass metric for each server based on that server’s set of snapshots.

The update process may further comprise storing the snapshots of the slave server that were taken after the common snapshot.

According to a third aspect of the invention there is provided a system for replicating a filesystem between a first server and a second server prior to and following a partition between the first server and the second server, the system comprising: snapshotting means for taking snapshots of a current state of the filesystem on the first server at predetermined points in time following modification of the filesystem, each snapshot recording differences between the current state of the filesystem on the server and the state of the filesystem on the server at the time point of a previous snapshot; replicator means for continually replicating the snapshots taken on the first server to the second server as soon as they are taken; detection means configured such that upon detection of a partition, both the first and the second server become masters for the filesystem and accept new modifications to the filesystems; updating means configured to perform an update process to update the filesystem after recovery of the partition, the update process comprising: identifying which of the first server and the second server contains the most current version of the filesystem; nominating the server so identified as the master server and the other server as the slave server; identifying a snapshot that is common to both the master server and the slave server; and replicating subsequent snapshots from the master server to the slave server.

Identifying which of the first server and the second server contains the most current version of the filesystem may comprise calculating a centre of mass metric for the version of the filesystem on each of the servers, the centre of mass metric representing the average age of the snapshots of the filesystem on each server and the number of changes to the filesystem represented by the snapshots on each server.
Identifying which of the first server and the second server contains the most current version of the filesystem may further comprise identifying a set of snapshots of the filesystem that for each server, each set of snapshots containing snapshots only present on that server, and calculating the centre of mass metric for each server based on that server's set of snapshots.

The update process may further comprise storing the snapshots of the slave server that were taken after the common snapshot.

The system may further comprise storage means for storing the snapshots taken of the filesystem such that a previous snapshot of the filesystem can be selected by a user from the stored snapshots to restore the system to its state at the time of the selected snapshot.

The previous snapshot of the filesystem may be selectable by means of a user interface presented to the user.

According to a fourth aspect of the invention there is provided computer software which, when executed by appropriate processing means, causes the processing means to implement the systems and methods of the first, second and third aspects of the invention.
Brief Description of the Diagrams

Embodiments of the invention will now be described, strictly by way of example only, with reference to the accompanying diagrams, of which:

Diagram 1 is a schematic representation of a network cluster containing a first server A replicating a snapshot 2 to a second server B, which already has a snapshot 1;

Diagram 2 is a schematic representation of the network cluster of Diagram 1 with a network partition and a divergence;

Diagram 3 is a schematic representation of an example replication system configuration where there are three servers A, B and C and two filesystems, F and G;

Diagram 4 is a schematic diagram showing snapshotting replicator states;

Diagram 5 is a schematic diagram showing per-slave sending replicator states;

Diagram 6 is a schematic diagram showing receiving replicator states;

Diagram 7 is a schematic diagram showing controller state transitions; and

Diagram 8 is a schematic representation of a distributed protocol handler.

Diagram 9 is a schematic representation of the live-migration state machine transitions
Description of the Embodiments

Terminology
In this description we make use of basic mathematical symbols, including:

Defining objects:
\[ A := \text{definition} \]

Sets:
\[ \{1, 2, 3\} \text{ for unique elements 1, 2, 3} \]

Mappings:
\[ [1 \rightarrow A, 2 \rightarrow B] \text{ for unique keys 1, 2} \]

Ordered tuples:
\[ (1, 2, B) \text{ which may be of differing types} \]

Compound type definitions (named tuples):
\[ \text{Type}(A, B, C) \]

Assumptions
We assume the existence of two underlying systems which the present invention depends on:

1. A local filesystem which can contain arbitrarily many sub-filesystems (one for each application or database). Each filesystem can have arbitrarily many consistent point-in-time snapshots, each named with a locally unique string, and furthermore there is a mechanism to replicate the difference between two snapshots from one machine to another. Strictly by way of example, on server A there might be filesystem X which has snapshots \{1, 2, 3\} and on server B there might be snapshots \{1, 2\} of the same filesystem. The filesystem allows us to
replicate the difference between snapshot 2 and 3 (e.g. only the blocks on disk which have changed) to bring server B up to date so that it contains snapshots \{1, 2, 3\}. One example of a filesystem which satisfies these requirements is the open-source ZFS filesystem.

2. A group messaging service which allows messages to be sent between servers in the cluster. Crucially, the group messaging service provides certain guarantees about messages which are broadcast to the group: even over lossy, high-latency network links, message delivery is guaranteed to all currently active members of the group, and message ordering is logically consistent across all servers. Strictly by way of example, if server A sends one message to the group, and simultaneously server B sends another message, all the members of the group, including A and B will receive the messages in the same order. One example of a group messaging system which satisfies these requirements is the open-source Spread Toolkit.

Overview

During normal operation, the system will elect one master for each application, so that each application is hosted on precisely one server in the cluster. Changes which occur to that application on the master will be asynchronously replicated to \( n \) slaves for that filesystem for \( n + 1 \) total copies of the filesystem. This makes the system \( n \) redundant as it can tolerate the failure of \( n \) servers.

Replication system

**SnapshotGraphForest data structure**

Fundamental to the cluster's ability to perform data replication between servers under arbitrary failure and partition conditions is the **SnapshotGraphForest**. This data structure represents the global state of a given filesystem across all servers in the cluster.

We begin with the simple case of a cluster with one filesystem F.
Diagram 1 represents server A replicating snapshot 2 to server B, which already has snapshot 1.

A snapshot graph forest is a set of snapshot graphs. A snapshot graph is a directed acyclic graph (DAG) of snapshot nodes. A snapshot node is a specific, globally unique version of a filesystem, including a set of parent edges, which identifies that snapshot node’s position in the graph.

The graph is a DAG because a snapshot node can have multiple parents and also multiple children. It’s acyclic because parent snapshots are always older than child snapshots, so a cycle in the graph can never be formed.

Each snapshot node is defined by the type SnapNode(id, [id_p → (srvs, count, imm)]) where id is the globally unique snapshot identifier, id_p is the parent pointer which refers to the id of the earlier snapshot on a specific server on which this snapshot is held (this may be NULL if it is the first snapshot, in which case it is said to be based on the origin), srvs is the set of servers on which the snapshot is presently stored, count represents the number of filesystem modifications captured by the snapshot with respect to its parent snapshot, and imm represents whether the given snapshot is immutable (whether it may be deleted) on the given server. We will ignore imm until we discuss pruning later.

Observe that a SnapNode object can represent the state of the filesystem on multiple servers at once, and capture the fact that on different servers, the parent snapshot of each snapshot may differ, even though the data the snapshot captures is identical.

A snapshot graph is defined as SnapGraph(set of snapshot nodes) where all the snapshot nodes in a graph are reachable via the parent and child pointers of those nodes.

In the example in diagram 1, before the replication, there is a graph G in the forest:

\[
G := \text{SnapGraph}((\text{SnapNode}(1, [\text{NULL} \rightarrow \{\text{A,B}\}, 2])), \\
\quad \text{SnapNode}(2, [1 \rightarrow \{\text{A}\}, 3]))
\]

Snapshot 1 is an initial snapshot which is stored on both A and B with two changes recorded between the origin and the snapshot, and snapshot 2 is based on (has a parent of) snapshot 1 and has a copy only on server A.
The complete snapshot graph forest for this configuration is \( \text{SnapForest}([G]) \). That is, there is only one graph \( G \) in this forest (there are no completely disconnected sets of snapshot nodes, or, all nodes are connected to all other nodes).

After the replication of snapshot 2 onto B, the graph \( G' \) has the new state:

\[
G' := \text{SnapGraph}(
\begin{align*}
\text{SnapNode}(1, [\text{NULL} \rightarrow ([A, B], 2)]), \\
\text{SnapNode}(2, [1 \rightarrow ([A, B], 3)])
\end{align*}
\)

Note that B now has a copy of snapshot 2, indicated in bold above.

**Diverged graphs**

Consider that a cluster may become partitioned from a group \{a_1, .. a_m, a_{m+1}, .., a_n\} for \( n > m \), into two groups L: \{a_1, .. a_m\}, R: \{a_{m+1}, .. a_n\}. In fact a failure may cause arbitrarily many partitions, but we describe the two-partition case, which generalises to arbitrarily many partitions.

Observe in fact that that all failures can be generalised to partitions, for example the failure of a single server \( a_i \) can be considered as the partition into the groups \( \{a_j \mid j \neq i\} \) and \( \{a_i\} \). The failure of a network switch can be considered as a partition into \textit{num-ports} many groups each containing a single server.

During a partition, all sides of a partition elect new masters for all available filesystems. Now the data on both sides of the partition may begin to diverge as changes get made to the filesystems on both sides of the partition.

Diagram 2 shows the same cluster as before but with a network partition. Now servers A and B cannot talk to each other, and so they both elect themselves as the new master for the filesystem in question. Both servers then might observe modifications to their filesystem F and server A might take snapshot 3, which captures 1 modification, and server B might take snapshot 3' which captures 4 modifications.
The global state of the SnapShotGraphForest for this system is now:

\[
G := \text{SnapGraph}\left(\begin{array}{l}
\text{SnapNode}(1, [\text{NULL} \rightarrow \{A, B\}, 2]), \\
\text{SnapNode}(2, [1 \rightarrow \{A, B\}, 3]), \\
\text{SnapNode}(3, [2 \rightarrow \{A\}, 1]), \\
\text{SnapNode}(3', [2 \rightarrow \{B\}, 4])
\end{array}\right)
\]

That is, there are now four SnapNode objects, one for each distinct filesystem state captured by the system. Since snapshots 3 and 3' both have snapshot 2 as a parent, the filesystem state is said to have \textit{diverged}. Note that only after the network partition is recovered and A and B can communicate again that they can discover this complete graph by sending messages which include their filesystem state.

We will now consider one final example which demonstrates why it might be necessary to be able to express a forest of completely disconnected graphs. Suppose servers A and B remain disconnected and users on both sides of the partitions happen to add a filesystem G with the same name on both sides of the partition. Suppose then the system takes initial snapshots:

\[
S_1 := \text{SnapNode}(1, [\text{NULL} \rightarrow (A, 2)]) \quad \text{on A's side of the partition} \\
S_2 := \text{SnapNode}(1', [\text{NULL} \rightarrow (B, 3)]) \quad \text{on B's side of the partition}
\]

Now the resulting snapshot graphs will not be connected, and therefore the forest contains two disconnected graphs:

\[
\text{SnapForest}\{\text{SnapGraph}\{S_1\}, \text{SnapGraph}\{S_2\}\}
\]

Multiple graphs can also be caused by one server A being offline for long enough that the other server B has deleted all the common snapshots of the filesystem by the time A comes back online.
Sometimes it is useful to refer to a local forest which contains only information about a filesystem on a specific server. Observe that a local forest is always a forest which contains a single linear graph with no divergences because a filesystem on a single server must always have a linear structure of snapshots from the earliest to the latest.

Finally a note on the snapshot identifiers (id). These are defined as tuples SnapId(timestamp, server) where the timestamp is the number of milliseconds since the UNIX epoch and the server is the globally unique primary IP address of the server which took the snapshot. Note the distinction between the SnapId's server field which describes where the snapshot was originally taken, and the SnapNode's srvs field which indicates where copies of the snapshot are presently stored.

**Exploring a SnapshotGraphForest: calculating divergences, heads, centre of mass, candidate masters, and finding updates**

Given a global snapshot graph forest representing the present global state of a filesystem on a cluster, the aim of the system is to perform operations on the local filesystems on each server in the system in order to return to a globally consistent state where replication of changes from master to slaves may continue.

The operations which we can perform on filesystems (known as manipulator operations) are:

1. **Snapshot**: Take a new snapshot.

2. **Send**: Send incremental snapshot(s) or a complete filesystem from one server to another.

3. **Stash**: Stash (save) snapshots from a given “slice point” to a local stashing area.

4. **Prune**: Prune (delete) a snapshot.
Here we describe a process which can be used for detecting divergences and deciding which actions to perform.

First we define a traversal function, which, given a starting node (a snapshot identifier), visits each SnapshotNode in its connected graph via its parent and (deduced) child pointers. It constructs mappings of child and parent pointers and then performs a search of the graph accessible from the starting node, remembering which snapshots it has already seen to avoid loops.

From this we can define a graphs function, which given a set of SnapNode objects, removes a SnapNode from the set and adds its complete graph to the set of graphs until there are no SnapNode objects remaining, thereby taking an unstructured set of snapshot nodes to a set of snapshot graphs by establishing which nodes are interconnected.

Now we can define a heads function, to calculate which snapshots in a given graph are the competing most recent versions of each filesystem, the “heads” of the divergences. Given a graph as calculated by graphs, the heads of that graph are precisely the elements of the graph which have zero children in the graph.

We can define a restricted graph with respect to a server as the set of snapshot nodes restricted to the snapshots which have a copy on a given server. So in diagram 2, the complete graph is \{1, 2, 3, 3'\} but the graph restricted to server A is \{1, 2, 3\} and the graph restricted to B is \{1, 2, 3'\}. Note that snapshot nodes in a restricted graphs only ever have one parent edge.

Now we can define a centreOfMass function on a restricted graph, which calculates a weighted sum: a time-like value which is the average timestamp of all of the snapshots in the restricted graph, weighted by the number of modifications in that node’s single parent edge. Intuitively, a graph with a more recent centre of mass is more valuable than a graph with an older centre of mass, because the more recent centre of mass corresponds to more recent and more significant changes.

This is the formula which can be used to calculate the centreOfMass of a graph G restricted to a server A:
\[
\text{tail}(G|_A) = \{ g \in G|_A : g \neq \text{first}(G|_A) \}
\]
\[
\text{centreOfMass}(G|_A) = \frac{\sum_{g \in \text{tail}(G|_A)} \text{weight}(g) \times \{\text{time}(g) + \text{time}(\text{parent}(g))\} \times \frac{1}{2}}{\sum_{g \in \text{tail}(G|_A)} \text{weight}(g)}
\]

First we define the tail of a restricted graph simply as all the snapshots in that graph which are not the first snapshot. This is because the midpoint of each snapshot node \( g \) and its parent is only defined when \( \text{parent}(g) \) is not the origin. Then we can define the \textit{centreOfMass} of a restricted graph as the sum over the snapshots in the tail of the graph of the midpoint in time of that snapshot and its parent, weighted by the weight of each snapshot (number of changes between that snapshot and its immediate parent), divided by the total weight of the tail of the graph.

By way of example, consider which of the restricted graphs in diagram 2 have the highest centre of mass: the graph restricted to A has centreOfMass \((3\times(2+1)\times0.5 + 1\times(3+2)\times0.5)/(3+1) = 1.75\) whereas the graph restricted to B has centreOfMass \((3\times(2+1)\times0.5 + 4\times(3+2)\times0.5)/(3+4) = 2.071\). Intuitively, the graph restricted to B wins, and B should be elected the new master (because its data captures a greater weight of recent changes). Note that we do not count the weights between snapshot 1 and the origin, but this does not matter as it is equal in both cases.

To formalise this intuition, we define a \textit{chooseCandidateMasters} function which allows the system to handle the case where two or more servers have become competing masters for a filesystem due to a network partition. When the network partition recovers, the servers observe that they are in competition by exchanging lists of which filesystems each server thinks it is the master for, and which they are not (called a \textit{current masters} message) and furthermore they exchange the snapshot data necessary to construct the global forests to decide which server should continue to be the master.

The \textit{chooseCandidateMasters} function operates as follows: given a graph, it calculates the set of servers which are involved in the graph (i.e. which have a copy of any snapshot
node in the graph), and for each such server, calculates the restricted graph for that server. For each restricted graph, it calculates the centre of mass of that restricted graph, and finally it returns the set of servers which tie at the maximum centre of mass.

When the servers detect that both of them are currently the master, by inspection of their current masters messages, they both run the chooseCandidateMasters function based on the globally synchronised snapshot data; whichever server discovers that it is the best candidate master asserts ownership of the site and the other servers cede to the new master. If they tie, one is elected at random by the server with the lexicographically lowest IP address.

If a master observes that a slave has a completely disconnected (separate graph), it compares the weights of the disconnected segments, and the winning side instructs the losing side to completely stash this entire filesystem so that replication can begin from scratch.

Now we can define a process findUpdates which, given as arguments 1. a server which has been elected as master, 2. a complete SnapshotGraphForest for that filesystem's global state, and 3. a list of slave server names, decides which actions to take in order to resolve divergences and allow normal replication to continue on those slaves. The findUpdates function works by using the traverse function to start at the current master's most recent snapshot id (master_head), working backwards visiting each (parent, child) pair. As soon as it finds a common snapshot with any slave, it knows that the parent is the "slice point" for that slave, so it records the update slave \rightarrow (snapshot_id, master_head).

The output of findUpdates therefore is a set of replication actions:

\{slave \rightarrow (start\_snapshot\_id, end\_snapshot\_id)\}

This corresponds to the actions needed to be taken to bring the slaves (machines which have any copy of a filesystem with the same name, and which may have some common snapshots on which base a replication) up to date with the master, possibly resulting in the slaves needing to stash some data in case their data was diverged, in which case the start\_snapshot\_id corresponds to a non-head snapshot on the slave. Otherwise, it is the
most recent ("head") snapshot on the slave, and the replication event is known as a simple “fast forward” update.

The starting and ending snapshot nodes can be more than one arc apart from each other in the graph because the underlying filesystem is capable of sending more than one snapshot in a single replication event.

In the unlikely case that there are no divergences but the given master has an earlier head snapshot than the slave, (i.e. the snapshots on the slave up to the first common snapshot are a strict superset of the snapshots on the master) the master is respected and the slave is instructed to stash the filesystem up to the point at which the master can continue replicating. This special case is expressed as an update where start_snapshot_id and end_snapshot_id are identical. This should not occur in practice.

The master runs the findUpdates function and sends the result, for each slave, as an instruction (a replicate message) to the slave to begin a replication. Now we will cover the details of how the replication proceeds in terms of state transitions between the participating components on the master and its slaves.

Stashed data may optionally be offered to the user in case they wish to recover data from the losing side of the partition.

**Replicators**

**Overview**

There are five types of objects which participate in the mounting, unmounting and snapshotting of filesystems, replication of data to slaves, and pruning of snapshots:

1. Controllers, of which there is exactly one per server. A controller is responsible for synchronising global state across all the servers, electing masters, adding and removing slaves, and brokering communication between the state machines and the group messaging protocol. It also implements load-balancing in terms of live-migration.
2. Mount handling replicators, which handle safely mounting and unmounting a filesystem. These exist on both masters and slaves, one per filesystem.

3. Snapshotting replicators, which exist on a master (one per filesystem), which receives notifications that a filesystem has been modified and decides when to take new snapshots.

4. Per-slave sending replicators, which exist on a master (one per slave per filesystem) and which communicate over the group messaging protocol to the receiving replicators (via the controller) in order to mediate the transmission of snapshot data from master to slave according to the results from the SnapshotGraph `findUpdates` function.

5. Receiving replicators, which communicate with per-slave sending replicators to mediate the receipt of snapshot data from master to slave.

Diagram 3 shows a one possible configuration where there are three servers A, B and C and two filesystems, F and G. This diagram corresponds to a `currentMasters` mapping of:

\[
\begin{align*}
F & \rightarrow \text{server A}, \\
G & \rightarrow \text{server B}
\end{align*}
\]

In this example, Server A is the master for filesystem F and server B is the master for filesystem G. Server C is a slave for both filesystems, and the cluster is configured to replicate filesystem data to two slaves per filesystem. The heavy lines in diagram 3 represent the flow of filesystem snapshot data.

**Controllers and mount handlers**

Each controller has a filesystem mount handler per filesystem, and each filesystem mount handler is in one of two states, RECEIVING or SENDING. If a mount handler is in RECEIVING, its filesystem is unmounted and it has a receiving replicator. If a mount handler is in SENDING, its filesystem is mounted and it has a sending replicator. Changes are actively made by the application to that filesystem, snapshots are made of it
by the snapshotting replicator, and the sending replicator's per-slave replicators, one per
slave, are responsible for sending the snapshots to the waiting receiver.

The heavy lines in diagrams 4, 5 and 6 correspond to the usual success cases, other lines
corresponding to error-handling or partition-recovery states.

**Snapshotting replicator states**

See diagram 4.

A snapshotting replicator receives notification of filesystem modifications and schedules
snapshots to be taken. When the snapshots are taken it informs its per-slave sending
replicators that they should check whether to initiate an replication event to its slave,
which has a receiving replicator set up ready to receive.

It begins in a LOADING state, which means it is interrogating the filesystem for current
snapshot state and loading it into its forest. When this finishes, it enters a READY state.
When it reaches the READY state, it informs the controller of the new state, which the
controller broadcasts to other nodes in the cluster. When a scheduled snapshot is due to
occur, it enters SNAPSHOTTING for the duration of the snapshot taking place.

It maintains a *global forest* which represents the global state of the snapshot data on all
nodes for that filesystem. It is informed about the other servers' state by an
`informGlobalState` interface which its controller calls when it receives updates about
global state from other servers in the cluster.

The scheduling of snapshots in response to *modified* notifications works as follows:

- If a filesystem receives just one modification, it is snapshotted within a
  `SNAPSHOT_QUICK` timeout.
- If a filesystem receives many modifications within a `SNAPSHOT_QUICK`
  interval, it takes a snapshot at the `SNAPSHOT_INTERVAL` timeout, which is
  longer.
This means that if a filesystem is modified heavily, it gets snapshotted every \texttt{SNAPSHOT\_INTERVAL} seconds, whereas if it is just modified once, it gets snapshotted within \texttt{SNAPSHOT\_QUICK} seconds. Some sample values of these values are 30 seconds and 60 seconds, respectively.

When a snapshot is complete, this state machine also handles pruning asynchronously, in order to keep the number of snapshots to a reasonable number (typically around 100 per filesystem). Pruning is described in detail later.

**Snapshottimg databases**

Snapshottimg databases will require co-operation from the database in order to coerce it into making its on-disk state consistent by holding a lock on the database during the snapshot operation. In one aspect, the invention achieves this by issuing a "\texttt{FLUSH TABLES WITH READ LOCK}" query to a MySQL database. Other database engines can be integrated with the invention with equivalent mechanisms. This allows databases, as well as applications and mailboxes to be snapshotted, automatically recovered and live-migrated between servers. Databases and related filesystem snapshots may be co-ordinated in time such that an application's state on the disk and in the database is consistent.

**Per-slave sending replicator states**

See diagram 5.

A per-slave sending replicator is responsible for initiating replication events in conjunction with a remote receiving replicator. It begins in the \texttt{READY} state (no loading is necessary because it refers to the forest of its parent snapshottting replicator). When it has \texttt{check} called on it, either because a new snapshot has been created, or a server has just been added as a slave and a new per-slave sending replicator created for it, it calls \texttt{findUpdates} on its forest.
When `findUpdates` indicates that a specific data stream (with defined start and end snapshot ids) should be sent from the local machine to the remote slave which the per-slave is set up for, it sends a message over the group messaging protocol to the remote receiving replicator and goes into state SENDING_WAITING. If the remote receiving replicator accepts the replication attempt, the per-slave sending replicator goes into state SENDING_RUNNING and the snapshot data begins to flow over the network. When all the snapshot data has been sent, the snapshotting sending replicator enters the WAIT_FOR_ACK state, which means it is waiting for the remote receiving replicator to acknowledge correct receipt and storage of the data indicated. When that happens (again via the group messaging protocol), the per-slave sending replicator re-enters the READY state.

If at any point a failure message is received from the remote side, or if a timeout fires (which may occur if the remote machine fails or the network becomes partitioned), the state machine transitions to PAUSE and then transitions back to READY after a further timeout. This allows replication to continue, without causing large numbers of messages to be sent in case the remote side is temporarily unable to receive new replication events.

**Receiving replicator states**

See diagram 6.

When a server is a slave for a filesystem, the filesystem mount handler is in RECEIVING mode and has ensured that the filesystem itself is unmounted, and available to receive filesystem updates from a remote per-slave sending replicator (of which there will usually be exactly one, since there is only one master per filesystem within any given network partition – if there is more than one master after a network partition and subsequent recovery, the master negotiation described above will ensure that one master cedes in a short amount of time so that replication can continue).

The receiving replicator starts in the LOADING state, where it is interrogating the filesystem for current snapshot data. When it receives the filesystem data, it informs its
controller of the current snapshot state. The controller informs other servers in the cluster of this, and the receiving replicator enters the READY state. Having informed other servers of the current state, they may decide, based on their global forests calculations, that the slave has diverged, or that it needs a simple "fast-forward" update.

If the update is a fast-forward update, the replicator proceeds directly to the RECEIVING state, and snapshot data flows over the network. When it completes transitions to the LOADING state, checks that the expected data was received correctly, then initiates asynchronous pruning and immediately becomes ready for the next replication event.

If the update is not a fast-forward update, the replicator instead transitions into the STASHING state, where it stores in a local "stash directory" binary copies of the snapshots between the "slice point" (the end_snapshot indicated by the sending replicator which is the latest common snapshot between the master and the slave) and the current head of the filesystem on the slave. Once this stashing is complete, the filesystem is immediately ready to receive the changes and replication proceeds as normal. The start snapshot is then marked as immutable so that the stashing process can be reversed.

In some situations the local filesystem on the slave can be modified (even though it is meant to be unmounted, administrators may accidentally mount it and modify it, for example). In this case, the replication will fail, however the receiving replicator detects this case and transitions into LOCAL_MODS, which causes the local modifications to be snapshotted and safely stashed. The receiving replicator emits a failure message and the per-slave sender will transition to PAUSE and try again when its timeout fires, so that replication can continue.

**Pruning algorithm**

The processes above describe creating snapshots, but not destroying them. It's important to destroy old snapshots in order to bound the number of snapshots to a reasonable number. Filesystem operations become slow when you have more than a few hundred snapshots. To a user, the difference between two point-in-time snapshots taken a minute apart from over a year ago is likely to be less important than the difference between two
point-in-time snapshots from the last few minutes, so it makes sense to prune older snapshots more aggressively than newer ones. Pruning is the process of collapsing the changes from a number of sequential snapshots into a single snapshot.

An important property of the pruning process is that it results in the same snapshots being chosen for deletion on all the servers in the cluster. This is so that the findUpdates process will find a recent common snapshot and avoid sending unnecessarily large amounts of replication data.

The pruning algorithm works by defining a set of sections: typically the last hour, last day, last week and last month, and then "filling in the gaps" between the sections with "waypoints", for example the system can be configured so that all snapshots from the last 60 minutes will be kept, hourly snapshots are kept for the last day, daily snapshots are kept for the last week, etc.

Snapshots are suggested for deletion by the suggestedDeletions function if they are not the closest snapshot to a waypoint.

Because the waypoints are quite stable with respect to the passage of time, almost the same pruning decisions are taken on all servers, even if pruning occurs at slightly different times on different servers.

Very recent snapshots will also be excluded from consideration for deletion, and immutable snapshots are never deleted. Snapshots are marked immutable (locally on a specific server only) if a stash has happened which is based on that snapshot, since to recover a stash of a snapshot which is based on an intermediate snapshot, the intermediate snapshot must still exist, and therefore for the stashes to be usable to recover data from, snapshots which the stashes are based upon must be made immutable and never deleted until the stash is discarded.

Both the Snapshotting replicator and the Receiving replicator utilise this pruning algorithm to keep the number of snapshots on masters and slaves within reasonable bounds.
The system may optionally expose an interface for users to roll back to specific snapshots, clone new applications and databases from snapshots at a given point, and to manually set certain snapshots to be immutable.

**The Controller**

This section explains the overall “controller” process which is responsible for being aware of which servers are online within the current network partition (if any) and therefore which server should be elected as the master for each site. It is also responsible for adding slaves if a filesystem is *under-replicated* and removing slaves if a filesystem is *over-replicated*.

**Cluster boot and merging process**

During normal operation, servers will broadcast several messages over the group messaging system at appropriate intervals:

1. *Heartbeat* messages – asserting the liveness of each server, and that each server is passing its own self test (that all systems and processes are operating correctly on that server). This data is stored on every machine in a mapping called the *liveness mapping*.

2. *Available data* messages – stating which snapshots of which filesystems each server has, used to determine the filesystem state and to inform the replication decisions as described. This data is stored on every machine in a mapping called the *available data mapping*.

3. *Current masters* messages – stating which servers are currently master for which filesystems. This data is stored on every machine in a mapping called the *current masters mapping*.

4. *Load value* messages – stating the amount of load currently being generated by each application on each server, used in the load balancing calculations.

There are also a number of periodic checks which may run at configured intervals:
1. Emit heartbeats
2. Emit current masters messages
3. Checking dead filesystems
4. Checking load balancing
5. Checking redundancy (over/under-replication)

When a server starts, it begins by reading the current filesystem and snapshot state. If there was a clean shutdown last time, it may read this data from a local cache file which also includes data regarding the previous current masters state and also the servers which were live just before this server was previously shut down (a CHECK_TIMEOUT grace period is applied for each server which was previously live to come back online before the controller “rescues” their sites). This is to facilitate quick cluster restarts when necessary, because excessive remounting, which is slow, is avoided.

**Heartbeat messages**

The controller uses the group messaging system to emit a heartbeat from each server each second. The system records the last time it heard from each server and every server can therefore detect which servers are live (i.e. in the same partition as it) based on a CHECK_TIMEOUT interval, and which servers are silent (failed or partitioned).

**Avoiding premature actions**

When a server is starting up, it may observe some state which appears to indicate that it should perform some action, such as rescuing apparently dead filesystems. However this behaviour may be wholly incorrect, because it may not have yet heard all of the information it needs in order to make the correct decision. Therefore, we defined a concept called heardFromAllServers, which defines that the set of live servers (servers from which we have heard a heartbeat in the last CHECK_TIMEOUT seconds) must be a subset of the keys of the mapping in question. Therefore we guard the periodic checks which would perform such potentially damaging actions with a heardFromAllServers
check, checking either that we have heard available data or current masters messages from all servers.

Diagram 7 describes, therefore, the states which a server will go through when it starts up, and how a new server joining, emitting a heartbeat, but not yet having asserted its ownership of filesystems can cause the other servers in the cluster to delay running their loops again until the new server has emitted a datasets message. Only when all servers have heard all other live servers emit a datasets message will any server be allowed to emit a current masters message, and only when there is global consensus on the current current masters state will any server therefore be able to run checkDeadSites. This makes the cluster very robust to servers or networks failing and being brought back online without making partially-informed decisions which could cause unfortunate consequences, such as an old server coming online and claiming to be the master for a large number of filesystems, when in fact it had two week old copies of all the data.

**Decision making using leaders**

The system defines a *leader for a filesystem* as the server with the lowest lexicographical IP address which has a copy of that filesystem. This is a simple way of breaking symmetry in a distributed system.

Note that being a *leader* for a filesystem is very different to being the *master* for it. The leadership check is only used in order to establish which server is able to make decisions about changing which server is the current master is for that filesystem. This mechanism stops multiple servers attempting conflicting migrations of filesystems simultaneously.

**Current masters message emits binary values to converge on global state consensus**

The current masters message contains, from each server, a list of which sites it is and is not hosting. This allows all servers to construct a globally consistent current masters mapping and to resolve competing masters after partition recovery.
It is upon receipt of an current masters message where the case of two competing masters in a recently-merged partition can be detected and handled. This is done by using the \texttt{chooseCandidateMasters} function described in the snapshot graph section.

The system broadcasts a binary value True or False for each filesystem. By looking at the totality of current masters messages from all servers, and comparing to the system's own current masters mapping, we correctly synchronise the global state using the following logic:

- IF the server is claiming to host the filesystem, but we do not think it is hosted there OR the server is claiming to not to host the filesystem but we think it is hosted there
- AND we are the leader for that filesystem
- THEN move it to the best server, based on the candidate masters calculation

**Local and remote redundancy calculations (addSlaves)**

The replication checking loop, for each filesystem a server is presently a master for, checks two things: whether a filesystem is under-replicated, in which case it calls \texttt{addSlaves} on the snapshotting replicator which creates some new per-slave replicators for the chosen new slave servers (which then automatically create new receiving replicators, and the filesystem gets copied to the new slaves).

The second check is whether a filesystem is over-replicated, in which case it issues a \texttt{deleteFileSystem} message, which causes the remote slaves to trash their copies of the filesystem, and the per-slave replicators are shut down.

The cluster is aware of which servers are in a local data centre and which servers are in a remote data centre. This allows it to be smarter about how many slaves in each locality to replicate to, based on the configuration of the cluster. For example, a cluster administrator can decide that she wishes to have a localRedundancy value of 2, which means two servers in the local data centre have each filesystem replicated to them in addition to the master (so that the cluster can cope with the failure of 2 local servers), a
globalRedundancy value of 1, which means that two other data centres (localities) must
have each filesystem replicated to them, and a slavesPerRemoteLocality value of 1,
which means that each remote locality must have one server which gains a copy of the
filesystem.

Since filesystems and applications may be live-migrated from one data centre to another,
additional replicas might be automatically created in the new data centre when the
filesystem arrives there, and some replicas in the old data centre might be removed.

Checking dead filesystems

If a server fails, some filesystems will cease to be hosted on any live server. In this case,
the checkDeadFilesystems loop on each server calculates the set of dead filesystems
which it can do anything about, its concerns: those filesystems which that server has a
copy of for which the current master of the filesystem (if any) is not presently live.

For each of these filesystems, each server ascertains whether or not it is the current leader
for the filesystem, and if it is, it elects a new master for the filesystem based on one of the
optimum servers from the chooseCandidateMasters function.

Distributed protocol handler

Mediating all protocol access (example protocols: HTTP, HTTPS, MySQL client
protocol, SMTP, POP and IMAP) between clients and the system is the distributed
protocol handler described in diagram 8.

They allow any request for any filesystem to be directed to any server in the cluster. This
means that, for example, a DNS configuration can be set up so that a website has multiple
'A' records, each pointing to different servers in the cluster, to take advantage of the
(limited) built-in redundancy in HTTP where a web browser will try an alternative 'A'
record if the first one it tries is unavailable.

On each server, the protocol handler “sits in front of” the actual application servers
(example application servers: Apache, MySQL server, Exim, Dovecot). In addition, the
protocol handler is connected to the controller described above, and has access to its *current masters mapping*. The protocol handler can “speak” just enough of each protocol to establish which filesystem the request should be routed towards. The example in diagram 8 shows a configuration of two servers where the request came to server A for filesystem F. The protocol handler chooses the backend by inspecting the controller’s *current masters mapping*, and discovers that it needs to route the request to server B, so its outgoing proxy connects to server B’s incoming proxy. Server B then inspects its *current masters mapping* (which is in agreement with server A’s by the global state consensus described above) and routes the request to its own “backend server”. At this point the connections are “seamlessly joined up” so that neither the client nor the backend server can tell that this is not a perfectly ordinary client connection. The client and the correct backend server then communicate as normal (for example: the server sends the client a web page over an HTTP connection), but simultaneously the protocol handlers are keeping track of the connection passing through them.

They need to keep track of the connection because they have the ability to pause new requests on demand. This is in order to implement seamless live-migration. If the controller has requested that a protocol handler pauses connections to a given server, it will, in one of two modes. It will wait a timeout for the “in-flight” connections to close naturally, while pausing all new incoming connections, then:

1. If the pause is *forced*, and if the current in-flight connections do not close naturally, it will forcefully terminate them.

2. If the pause is not *forced*, it will wait a timeout for the connections to die naturally, while pausing all new incoming connections. If the in-flight connections do not complete in the time allocated, the pause attempt is abandoned and the new paused connections are “unleashed”.

If the pause succeeded, it waits until the controller requests that the pause is “unblocked” at which point the system checks which backend should be connected to again by asking the controller (crucially, the backend may have changed during the pause operation), and connects to the potentially-different backend server, unleashing a “flood of requests”
which were building up during the pausing process onto the new server, which can then process them as usual. If the delay is sufficiently short, end users will only notice a small delay.

**Live-migration**

Now we have all the pieces of the puzzle to describe the complete live-migration process. To recap, we can:

- Ensure that replication proceeds to slave servers even under failure and partition conditions, and recover after the recovery of those conditions.

- Control in-bound connections with the distributed protocol handler so that any request can be addressed to any server in the system, and so that the system can momentarily pause in-bound connections, wait for in-flight ones to complete, and the redirect requests to a different server.

Now we can describe the live-migration state machine transitions and protocol. The controller may, under the user's direction or because of one of two mechanisms described below, choose to initiate a live-migration of an application from one server to another.

The controller creates a Thrower object in state INIT, which is responsible for simultaneously controlling the replication system and the distributed protocol handler. This Thrower object sends a requestmoveload message to the remote controller, which attempts to allocate a slot for the live-migration (there are a finite number of live migrations which are allowed to occur in parallel). If a slot is allocated, it creates a Catcher object in state INIT, and the catcher issues an acceptmoveload message. The Thrower then instructs its snapshotting replicator to construct a per-slave-replicator for the target server, in case it is not already a slave. The Thrower then sends a latestsnapshot message, which instructs the catcher to enter a PREREPLICATION state until that snapshot has been received. This may not be the final snapshot which is used in the replication, but it at least gets the catching server “quite up to date” so that the critical path element of the live-migration, where in-bound requests for the filesystem are momentarily blocked, is as short as possible. If the catcher observes that it already has
this snapshot, it can bypass the PREREPLICATION phase and initiate a continuemove-load message immediately. Otherwise, it emits a prereplication message and then when the catcher's replication system observes the snapshot arriving, it informs the thrower that it may continue by sending a continuemove-load message. The thrower then instructs its distributed protocol handler to begin pausing all new incoming requests and to notify it when all current in-flight requests are finished. The catcher does the same. Now the entire live-migration process can be in one of two modes, forced or unforced. If the mode is unforced, and there are long-lived connections to the current master (such as an IDLE IMAP connection, for example), the pausing can be abandoned which causes the entire live-migration to be abandoned (it can be useful, for example if it is necessary to completely shutdown a server, to force the live-migrations so that they always succeed in a short amount of time, at the cost of possibly closing some long-running connections). When both sides' distributed protocol handlers succeed in closing all current connections and pausing/blocking all new incoming connections, the thrower instructs its filesystem mount handler to unmount the filesystem, so that no further changes can possibly be made to it, at which point it takes a final snapshot of the filesystem and replicates this final snapshot to the catcher, all while new incoming requests for the application are paused. When the replication succeeds, the catcher mounts the filesystem, and emits a completemove-load message which results in both the thrower and the catcher unblocking their respective distributed protocol handler and so a flood of paused requests (users waiting patiently for the few seconds that this process takes) are unleashed on the new master for the site.

**Driving live-migration**

The controller has two mechanisms for automatically initiating live-migration events. These are a load-balancing mechanism and an application locality preference mechanism.

**Load balancing: load > av + Q**

All the servers in the cluster are constantly trading information about the current levels of load that are being generated by each application, for example by measuring the sum of
the total request times for that application within a ten second period. These measurements are “smoothed out” by using an exponential decay algorithm over 10 minutes (the same algorithm used by UNIX load average calculations). Servers are continually (in the checkLoadBalancing loop) checking whether their total load (the sum of the load across all of their applications) exceeds the average load in the cluster plus a “fudge factor” $Q$, which exists to stop the servers constantly trading load. If a server’s load exceeds $av + Q$ then the server elects a recipient server which is the server with the lowest load out of all the servers, and picks a site out of its current sites which is the maximally loaded site which will not cause the recipient to itself think it is over-loaded. This is known as the “anti-hot-potato choice function”, because it stops servers constantly trading load. The site which is chosen is live-migrated to the recipient.

The emergent behaviour from this simple set of rules is that servers will automatically load-balance themselves by migrating entire applications around between servers in the cluster. Furthermore, if one specific application gets a large spike in traffic, that application itself will not get live-migrated (because the anti-hot-potato-choice function forbids it); rather all the other applications on that server will get migrated away, leaving that server to be a dedicated server for that application.

**Application locality preferences**

Recall that the cluster may be distributed across geographically diverse regions. Users may wish to express a preference such that if a given region is available (if there are servers which are online there) then their site should be primarily hosted there. If the user specifies or changes this preference (which may be stored in a database), the controller detects the change and initiates a live-migration of both the application and any dependent databases. This is important so that applications and their databases are always stored in geographically local regions, since database access is often assumed to be low-latency. It may also be important for an application to not be hosted in or replicated to a given locality, in order to comply with local legislation.
Protecting Against User Error

In data protection systems that protect against hardware failure, such as RAID or synchronous replication, if a user accidentally deletes data the deletion is replicated to the replica device(s) and the deleted data will be permanently lost.

As is explained above, the system of the present invention continually takes point-in-time snapshots of all of the data stored on the system, and these snapshots are stored so that they can be accessed by a user, for example via a web interface which presents a graphical representation of the available snapshots. If the user accidentally deletes data from the system a previous data snapshot can be selected using the interface, by selecting one of the snapshots represented graphically, and the system can be restored or reverted to its state at the time at which the selected snapshot was taken, e.g. prior to the deletion, without requiring intervention by a system administrator.
CLAIMS

1. A system for dynamic migration of applications between servers, the system comprising a plurality of servers for hosting applications, each of the plurality of servers comprising a protocol handler for receiving requests for applications, wherein the protocol handler is configured to pause incoming requests for an application during migration of applications between servers.

2. A system according to claim 1 further comprising a load balancer for measuring load on one of the plurality of servers caused by one or more applications hosted on that server, the load balancer being configured to initiate migration of one or more applications from the measured server to another server when a predetermined load condition of the measured server is met.

3. A system according to claim 1 or claim 2 wherein the plurality of servers each has a controller that maintains a record of the server on which an application is currently hosted, and the protocol handler is configured to inspect the record to determine the server to which an incoming application request is to be directed.

4. A system according to any one of the preceding claims wherein the protocol handler is configured to pause incoming requests for an application and to terminate current requests for an application after a predetermined time period.

5. A system according to any one of claims 1 to 4 wherein the protocol handler is configured to pause incoming requests for an application for a predetermined time period and to release the paused requests if current requests for an application have not completed in the predetermined time period.
6. A method for replicating a filesystem between a first server and a second server prior to and following a partition between the first server and the second server, the method comprising:

at the first server, taking snapshots of a current state of the filesystem at predetermined points in time following modification of the filesystem, each snapshot recording differences between the current state of the filesystem on the server and the state of the filesystem on the server at the time point of a previous snapshot;

continually replicating the snapshots taken on the first server to the second server as soon as they are taken;

upon detection of a partition, both the first and the second server becoming masters for the filesystem and accepting new modifications to the filesystems;

after recovery of the partition, performing an update process to update the filesystem, the update process comprising:

identifying which of the first server and the second server contains the most current version of the filesystem;

nominating the server so identified as the master server and the other server as the slave server;

identifying a snapshot that is common to both the master server and the slave server; and

replicating subsequent snapshots from the master server to the slave server.

7. A method according to claim 6 wherein identifying which of the first server and the second server contains the most current version of the filesystem comprises calculating a centre of mass metric for the version of the filesystem on each of the servers, the centre of mass metric representing the average age of the snapshots of the filesystem on each server and the number of changes to the filesystem represented by the snapshots on each server.
8. A method according to claim 7 wherein identifying which of the first server and the second server contains the most current version of the filesystem further comprises identifying a set of snapshots of the filesystem that for each server, each set of snapshots containing snapshots only present on that server, and calculating the centre of mass metric for each server based on that server's set of snapshots.

9. A method according to any one of claims 6 to 8 wherein the update process further comprises storing the snapshots of the slave server that were taken after the common snapshot.

10. A system for replicating a filesystem between a first server and a second server prior to and following a partition between the first server and the second server, the system comprising:

    - snapshotting means for taking snapshots of a current state of the filesystem on the first server at predetermined points in time following modification of the filesystem, each snapshot recording differences between the current state of the filesystem on the server and the state of the filesystem on the server at the time point of a previous snapshot;

    - replicator means for continually replicating the snapshots taken on the first server to the second server as soon as they are taken;

    - detection means configured such that upon detection of a partition, both the first and the second server become masters for the filesystem and accept new modifications to the filesystems;

    - updating means configured to perform an update process to update the filesystem after recovery of the partition, the update process comprising:

        identifying which of the first server and the second server contains the most current version of the filesystem;
nominating the server so identified as the master server and the other server as the slave server;

identifying a snapshot that is common to both the master server and the slave server; and

replicating subsequent snapshots from the master server to the slave server.

11. A system according to claim 10 wherein identifying which of the first server and the second server contains the most current version of the filesystem comprises calculating a centre of mass metric for the version of the filesystem on each of the servers, the centre of mass metric representing the average age of the snapshots of the filesystem on each server and the number of changes to the filesystem represented by the snapshots on each server.

12. A system according to claim 11 wherein identifying which of the first server and the second server contains the most current version of the filesystem further comprises identifying a set of snapshots of the filesystem that for each server, each set of snapshots containing snapshots only present on that server, and calculating the centre of mass metric for each server based on that server’s set of snapshots.

13. A system according to any one of claims 10 to 12 wherein the update process further comprises storing the snapshots of the slave server that were taken after the common snapshot.

14. A system according to any one of claims 10 to 13 further comprising storage means for storing the snapshots taken of the filesystem such that a previous snapshot of the filesystem can be selected by a user from the stored snapshots to restore the system to its state at the time of the selected snapshot.

15. A system according to claim 14 wherein the previous snapshot of the filesystem can be selected by means of a user interface presented to the user.
16. Computer software which, when executed by appropriate processing means, causes the processing means to implement the systems or method of any of the preceding claims.
Patents Act 1977: Search Report under Section 17

Documents considered to be relevant:

<table>
<thead>
<tr>
<th>Category</th>
<th>Relevant to claims</th>
<th>Identity of document and passage or figure of particular relevance</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>1-3</td>
<td>WO 02/091179 A2 (SUN MICROSYSTEMS) Figure 10, pages 19-20</td>
</tr>
<tr>
<td>X</td>
<td>1-3</td>
<td>US 2006/0143350 A1 (MILOUSHEV et al) Paragraph 351</td>
</tr>
<tr>
<td>X</td>
<td>1-3</td>
<td>US 2006/0085789 A1 (LABORCZFalVI et al) Paragraph 321</td>
</tr>
<tr>
<td>X</td>
<td>1-3</td>
<td>US 2011/0047550 A1 (TACHIBANA et al) Paragraph 41</td>
</tr>
</tbody>
</table>

Categories:

X  Document indicating lack of novelty or inventive step
Y  Document indicating lack of inventive step if combined with one or more other documents of same category.
&  Member of the same patent family

A  Document indicating technological background and/or state of the art.
P  Document published on or after the declared priority date but before the filing date of this invention.
E  Patent document published on or after, but with priority date earlier than, the filing date of this application.

Field of Search:
Search of GB, EP, WO & US patent documents classified in the following areas of the UKC

Worldwide search of patent documents classified in the following areas of the IPC
G06F

The following online and other databases have been used in the preparation of this search report
WPI, EPDOC, TXTE, TXTT

International Classification:

<table>
<thead>
<tr>
<th>Subclass</th>
<th>Subgroup</th>
<th>Valid From</th>
</tr>
</thead>
<tbody>
<tr>
<td>G06F</td>
<td>0009/48</td>
<td>01/01/2006</td>
</tr>
<tr>
<td>G06F</td>
<td>0011/14</td>
<td>01/01/2006</td>
</tr>
</tbody>
</table>
# Patents Act 1977

## Further Search Report under Section 17

### Documents considered to be relevant:

<table>
<thead>
<tr>
<th>Category</th>
<th>Relevant to claims</th>
<th>Identity of document and passage or figure of particular relevance</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>6, 9, 10, 13-15</td>
<td>US 2006/0069890 A1 (COX et al) Paragraphs 241-247</td>
</tr>
</tbody>
</table>

### Categories:

- **X**: Document indicating lack of novelty or inventive step
- **Y**: Document indicating lack of inventive step if combined with one or more other documents of same category.
- **&**: Member of the same patent family
- **A**: Document indicating technological background and/or state of the art.
- **P**: Document published on or after the declared priority date but before the filing date of this invention.
- **E**: Patent document published on or after, but with priority date earlier than, the filing date of this application.

### Field of Search:

Search of GB, EP, WO & US patent documents classified in the following areas of the UKC:

- Worldwide search of patent documents classified in the following areas of the IPC:
  - G06F

The following online and other databases have been used in the preparation of this search report:

- WPI, EPDOC, INSPEC

### International Classification:

<table>
<thead>
<tr>
<th>Subclass</th>
<th>Subgroup</th>
<th>Valid From</th>
</tr>
</thead>
<tbody>
<tr>
<td>G06F</td>
<td>0009/48</td>
<td>01/01/2006</td>
</tr>
<tr>
<td>G06F</td>
<td>0011/14</td>
<td>01/01/2006</td>
</tr>
</tbody>
</table>