Seneca: remote mirroring done write

Minwen Ji, Alistair Veitch, John Wilkes
HP Laboratories, Palo Alto, CA
{minwen.ji, alistair.veitch, john.wilkes}@hp.com

Remote mirroring is widely deployed, but its complexities are surprisingly poorly understood. This means that data is less well protected than it ought to be, and possible optimizations are infrequently taken advantage of. To address these difficulties, we (1) demystify the design space by presenting a taxonomy of the approaches in use, (2) describe Seneca – a new asynchronous remote-mirroring protocol that supports write coalescing, asynchronous propagation, and in-order delivery, and (3) report on a performance and correctness validation of Seneca. We are confident that the result is a robust remote-mirroring protocol that provides good performance and predictable behavior in the face of a wide range of failure types, such as rolling disasters.

1 Introduction

Data mirroring is a classic technique for tolerating failures: by keeping two or more copies of important information, access can continue if one of them is lost or becomes unreachable. It is used inside disk arrays (where it is called RAID1), between disks or disk arrays, and across multiple sites, where it is called *remote mirroring*.

Remote mirroring is widely deployed whenever the cost of losing data matters. And it does matter: protection for information assets is often more important than for physical ones – at least the latter can be replaced after a loss. Gartner estimates that "Two out of five enterprises that experience a [site] disaster … go out of business within five years" [Witty2001]. Even lack of access to data is expensive: 25% of respondents to one survey estimated that outages cost them more than $250k/hour, with 4% estimating more than $5M/hour [EagleRock2001]. Remote mirroring can protect against both data loss and inaccessibility.

The design choices for remote mirroring are complicated by competing goals: keeping the copies as closely synchronized as possible, delaying foreground writes as little as possible, maintaining accessibility in the face of as many failure types as possible, and using as little expensive inter-site network bandwidth as possible.[1]

The basic trade-off is between better performance with lower cost against greater potential data loss, especially for recently-written data. Simple solutions, such as synchronously updating all copies, provide high resilience to data loss but have poor write performance and incur high network costs in remote-mirroring systems.

Contributions of this paper

Remote mirroring has been in use for quite a while, so it is usually thought to be well understood. Despite this, when we prepared a survey of the approaches used in practice, we found a wide variation in assumptions, techniques used, and the degree to which recovery is achievable. Additionally, many of the design choices are quite intricate and subtle, as we discovered when we ran our taxonomy past practitioners in the field. Our first contribution, then, is a taxonomy of the design choices for remote mirroring.

Our second contribution is the design of a robust remote mirroring protocol that provides resilience to many kinds and sequences of failures, low network bandwidth demands, and low (and tunable) data loss. We also look at its correctness, using an I/O automata-based simulation.

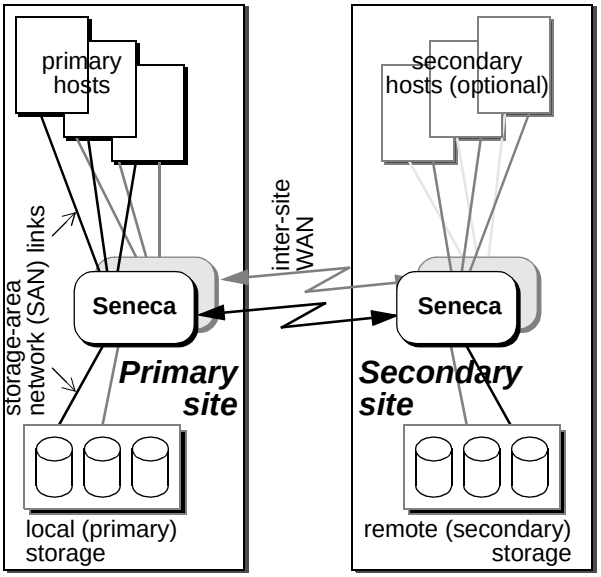


Figure 1: canonical remote mirroring system. The remote-mirroring function (labelled "Seneca") can be implemented in hosts, disk arrays, or in SAN appliances (as shown).

[1] For example, in South Carolina, the rental price of an 155Mb/s OC3 line in 2002 was about $460k/year [SC2002a]. This is equivalent to the capital depreciation cost of about 12TB of mirrored, enterprise-class (HP EVA) storage at late 2002 prices. Slower line costs are scaled accordingly: constant bit-rate ATM lines inside South Carolina are priced at an average of $170 per Mb/s per month [SC2002b]; variable-rate a bit less.

In California, SBC Pacific Bell [SBC2002] offered ATM service for:
DS1: 1.544Mbps = $750 per month plus one-time installation of $1200
DS3: 40Mbps = $5000 per month plus one-time installation of $3000
OC-3c: 148Mbps = $7000 per month plus one-time installation of $3000.

This 6:1 cost ratio is a result of the rapidly-changing telecom market, and is considered "not unusual" by our IT manager.

The final contribution is an evaluation of the protocol's performance, looking at how much it reduces network traffic using traces from real workloads.

2 A taxonomy for remote mirroring

The usual assessment criteria for remote mirroring designs include data accessibility (availability), resistance to loss or corruption of data (reliability), performance degradation in normal use, and the cost of operation, primarily in terms of inter-site network traffic. Accommodating these conflicting goals exposes many design and configuration options for remote-mirroring. The basic axes are as follows:

- the fault-coverage model;
- how closely synchronized the copies are;
- how updates are propagated;
- when updates are acknowledged;
- where the data duplication is performed.

All of these affect write performance and the amount of recently-written data that can be lost; some of them affect the amount of network traffic needed, too.

We use SCSI disk logical units (LUs) as the entities to mirror, because this is the most common practice. A SCSI disk drive exports a single LU; a disk array can have thousands of LUs. LU granularity allows different data to be given different degrees of protection: for example, a user file system may need less than a database index, which probably needs less than the data being indexed. The mirroring techniques described here can also be applied to other objects such as files, database tables, or object storage devices.

In what follows, a *local* site is one that is closer to the host and services data in normal operations; a *remote* site is the mirror of the local site; a *primary* site is one that actually services data; a *secondary* site is one that is not servicing data, either down or standing by as a backup. In the normal case, the local site is primary and the remote site is secondary.

2.1 Fault model

The fault model we used when designing the Seneca protocol is representative of those used in most remote mirroring designs – albeit more comprehensive than many of them. A remote mirroring system should tolerate failures of:

- host computers (hardware and software);
- links, switches, and hubs at each site (these comprise the local Storage Area Network, or SAN);
- wide- or metropolitan-area links between sites (WAN, MAN);
- any dedicated hardware used to implement remote mirroring;
- storage devices (but we ignore failures that are masked completely within a disk array);
- an entire site.

We assume fail-silent failures, and that recovery or repair of failed components is possible. A full-scale site disaster may take days to months to recover from, while a site power outage may be corrected in minutes, and a broken long-distance link may recover in a few seconds.

The traditional approach is to mirror storage across two sites, but more sites are possible, and may even be mandated soon for the financial services industry. We concentrate on the 2-site case here to simplify the exposition. The physical separation between sites is governed by the kinds of site failures that are to be tolerated. For example, a fire may take out a single building, a power outage all the buildings on a single campus, an earthquake or flood all the buildings within a metropolitan area.

Both repeated and multiple concurrent failures are expected. One failure can cause multiple components to fail (e.g., if it is in a shared component such as a power supply or air conditioning unit), or it can trigger a cascade of failures by increasing the stress on the rest of the system – including its operators.

We exclude certain scenarios: multiple concurrent site failures; pervasive software design or implementation faults, such as ones that fail to maintain duplicated copies correctly; and mis-installations. These may lead to a *disaster*, which we define as unacceptable data or availability loss, as may the occurrence of "too many" failures that occur before recovery actions from a previous failure can be completed. In all cases, data loss is less tolerable than lack of data availability.

2.2 Bounded divergence

Minimizing the risk of losing recently-written data means updating remote copies as rapidly as possible. This is easily achievable when the two copies are physically close (e.g., within the 10km single-link Fibre Channel distance limit), but becomes problematic if they are further apart, when the time to propagate an update across a long-distance link can be prohibitive. For example, the best-case speed-of-light round trip time across the continental USA is about 27ms, which is larger than a typical disk access, and huge compared to the access time to a disk array's cache. As a result, there are strong incentives to overlap the propagation with subsequent I/Os. The drawback is reduced reliability: writes that haven't been propagated to the remote site will be lost if the primary goes down.

Delayed propagation of updates can also reduce the worst-case long-distance network traffic needed because write-buffering allows bursty write traffic to be spread out more evenly over time [Ruemmler1993].

This leads us to the first part of our solution taxonomy: the amount by which the copies are allowed to diverge while the links between sites are up. The basic choices are "none" or "some". If the remote site becomes unreachable in the "some" case, or updates aren't propagating fast enough, there are two choices as to what to do at the primary: stall (or abort) writes until the secondary copy catches up, or switch to a mode with a looser divergence bound.

No divergence: all updates (writes) are propagated immediately to the remote site. This implies one of the *lock-step* protocols described below. Writes stall or fail if the remote copy is unreachable. *Analysis:* the safest thing to do – and the slowest. This is part of what most people mean when they talk about "synchronous mode"; it is the only mode in which non-catastrophic multiple failures will essentially never lose data: in all other modes (usually called *asynchronous remote mirroring*), data that hasn't propagated to other sites could be lost.

Operation- and/or **byte-count divergence:** the maximum divergence between the copies is deliberately limited to a small, fixed number of I/Os (e.g., one outstanding remote I/O per LU in EMC SRDF's "Semi-Synchronous" mode [EMC-SRDF]), or a bounded quantity of data. *Analysis:* this allows the transfer time to the remote site to be overlapped with a local I/O, which offers marginally better performance than the above. If the operation count or byte count is small, then the amount of data subject to loss is well bounded, so recovery may be simplified.

Resource-bounded divergence: the amount of divergence is bounded by some resource's size (e.g., the size of a log file or disk, or the size of an array's NVRAM cache, as in the HP XP1024 disk array [HP–XP1024, HP–XP–CA]). *Analysis:* even moderate amounts of divergence allows both good local performance and good inter-site link bandwidth needs. In practice, the limits are rarely met: disk array caches are often measured in gigabytes, and log files can easily be made much larger. Such systems still have to operate correctly when resource bounds are met, of course, and most do so by dropping into the unbounded divergence state discussed next. The order in which changes are propagated to the remote site matters, too – this aspect just captures how much divergence is allowed.

Unbounded divergence: there is no bound on the amount of divergence allowed. Faced with an unreachable site, and/or lack of log space for updates, this is all that can be done if local writes are to continue. It is common to keep track of which data has been updated (e.g., using a per-block, -track, or -cylinder data structure such as a bitmap) to reduce the amount of data that has to be sent over the link when the sites reconnect. *Analysis:* this allows the best performance, and the maximum amount of data-accessibility in the case of single failures: access is not denied unless the data is really unreachable. Once this state is entered, the only way to repair a remote copy is to propagate *all* the changes.

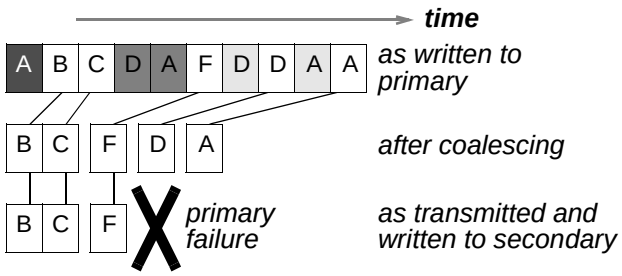


Figure 2: *rolling disaster from overwrites*. The top row shows a sequence of data blocks being updated; shading indicates generations of overwritten blocks. The second shows the data blocks left after coalescing. The third shows the state of updates after half the data has been transmitted; no updates to blocks A or D have been propagated, so the result is not consistent.

2.3 Single-LU propagation order

Write buffering enables *write coalescing* (or *overwrite absorption*): the overwriting of older writes in the queue of waiting data. This is most often used at the primary to reduce network traffic, but it could also be applied at the secondary to reduce the amount of work needed for an update. Even small amounts of memory for write coalescing can be quite effective [Ruemmler1993].

The primary effect of write coalescing on fault tolerance is to alter the order in which updates are applied at the secondary site. The goal is to ensure that the secondary is always at some *consistent state*—one that could be reached by a prefix of the sequence of writes applied at the primary. A failure part-way through applying a reordered set of updates can leave the remote site in a non-consistent state when it needs to take over from the primary. If the primary site is lost at this point, it may not be possible to recover the application that uses the data. (See Figure 2).

To simplify the discussion, we begin by considering the ordering options that apply to a single LU, and defer multiple-LU cases to section 2.4. The design choices are ordered by the amount of reordering, parallelism, and coalescing they permit.

Lock-step serial: at most one host write can be outstanding at a time; each copy is updated in a known order. This implies *full write-though* host acknowledgements – see section 2.5. *Analysis:* this provides the most careful form of dissemination for updates, and allows recovery from double outages, where the write order is otherwise not knowable. It is only appropriate when reliability is much more important than performance. The total time to perform the write is the sum of the response times of the writes to each copy.

Lock-step parallel: at most one host write can be outstanding at a time; each copy is updated simultaneously. This implies *full write-though* host acknowledgements – see section 2.5. *Analysis:* better performance than *lock-step serial*; the total time is the maximum of the individual response times, but a double-copy outage can leave the data in an uncertain state for writes that were in flight.

One of the lock-step algorithms is usually implied by the term "synchronous mode". Sometimes the secondary site is updated first (in case connection is lost, and the write should then be aborted); sometimes the primary. Combinations may also be useful, such as writing to the primary copy first, and then updating multiple secondary copies in parallel.

In-order asynchronous: here, the updates are propagated to, and applied at, the remote sites in the same order that they occurred at the primary. *Analysis:* this does nothing to reduce the amount of WAN traffic, although it can smooth out the traffic from write bursts if the divergence bound (i.e., delay pipeline) is large enough.

Dependency-preserving asynchronous: writes are provided with explicit dependency information, and propagation preserves these, but is otherwise free to reorder the updates. *Analysis:* this allows better performance by exploiting asynchrony where it would preserve the update semantics. We do not believe that this has been implemented for remote mirroring systems, probably because of the difficulty of providing the necessary dependency data, even though host-based variants have been shown to work well [Kondoff1988].

Write-coalescing batches with atomic update: this scheme explicitly delays sending a *batch* of updates to the remote site, in the hope that write coalescing will occur, and only one copy need be propagated. To avoid inconsistent states at the secondary, writes can be coalesced only within a batch, and batches must be committed atomically at the remote site: that is, all or none of the updates in the batch must be applied. *Analysis:* if overwrites are common, this can greatly reduce the amount of WAN traffic, at the expense of losing more updates if the primary site fails.

The size of the batch can be selected in several ways, such as the elapsed time, the number of updates, or the amount of data written, or bytes to transfer. It would be possible to select the batch size that achieves a particular target data-loss likelihood, taking the WAN link reliability into account (cf. AFRAID [Savage1996]).

Batches can be implemented by logs, or (at a coarser grain) with an array-based *LU snapshot* mechanism (e.g., EMC TimeFinder [EMC-TimeFinder], HP business copy XP [HP-XP-BC]). A snapshot is a virtual copy of an LU, and updates it, using copy-on-write, whenever the original LU is modified. Since the update traffic needed to implement a snapshot is primarily a function of the original LU's update rate and the snaphot's lifetime, not the total amount of data in the original LU, keeping a snapshot up to date is reasonably quick, and it is often possible to synchronize one in only a few minutes. (Some snapshot implementations take space proportional to the differences, so creating these is quick, too; some, like EMC TimeFinder copies, are complete mirror copies that can be incrementally updated: creating them is slow, but updating them can be done in time proportional to the amount by which they have diverged).

All atomic update schemes require sufficient buffering at the primary and the secondary to store the largest possible batch – which may be as large as the complete data size. Obviously, this isn't always possible (or cost-effective), so most schemes revert to non-atomic propagation when the divergence limit switches to unbounded.

Overwrite-log with atomic update: the overwrite-log scheme removes the hard send-batch boundaries in favor of a log of updates that allows write coalescing, together with the use of *receive batches* which are defined as the updates that occur between the first write of a data block and its overwrite, plus that overwrite. Receive batches represent the smallest unit that must be applied atomically at the secondary site. *Analysis:* this provides better behavior than the batched scheme, which wait until the end of the batch has been declared before they start propagating any data within the batch, perhaps delaying transmission longer than is needed, and unnecessarily exposing data to primary site failure. Seneca is the only example we know of.

Send barriers are used to mark the points beyond which write coalescing may not occur. These are needed to prevent the size of receive batches growing out of hand—but they need to be inserted only when necessary (e.g., to mark the end of a receive batch that has commenced transmission). For example, one simple dynamic scheme is to start transmitting pending updates immediately if the WAN is lightly used, or to artificially delay updates if the WAN is busy, to increase the overwrite rate.

Out-of-order asynchronous: the order in which updates are propagated to the remote site is unrelated to the order in which they occurred (e.g., a bitmap of updated tracks is kept, and the updates are propagated in track-number order). This case is implied if the primary is forced into the unbounded divergence case. *Analysis:* good performance, but a rolling disaster (see section 2.4) can occur if the primary site is lost before the secondary site has been fully updated.

Complete copy replacement: the old copy is simply overwritten in its entirety. *Analysis:* this may make sense if the number of changes to be applied is large (such as when a mirror copy was detached, most of the primary was updated, and now the secondary is being brought back into synchrony – a process sometimes called *resilvering*); if a partial failure

during the resynchronization is immaterial so the copy can simply be restarted to recover (e.g., if the copy is only being used to make a backup from); or if the expense of maintaining state about the differences is too high.

Explicitly delayed: instead of propagating changes as quickly as possible to a remote site, they can be explicitly delayed by at least a certain period of time, or until some event has occurred, such as a sanity check. For example: "keep 3 copies: one is the master; one is a remote copy that is as up to date as possible; and one is always as close to 12 hours behind the master as possible." *Analysis:* this technique can be used to allow time to run sanity checks (e.g., virus scans) over the data before installing it at the secondary.

Explicit delays might be used to replace backups, which are necessary today to guard against data corruption and operator error, both failures that mirroring does nothing about. (Indeed, most mirroring schemes try hard to propagate errors at the speed of light!)

2.4 Multi-LU propagation order

If an application updates two LUs, on the same or different arrays, and there are consistency requirements between those updates, then additional steps have to be taken to handle these needs. It is common to use a *consistency group* to specify the LUs involved. The important thing is the degree of guarantee offered by the consistency group:

No inter-LU consistency. *Analysis:* the name *rolling disaster* says it all—and comes from a sequence of storage device failures at different times (e.g., as the result of a fire or flood in a data center), or from the case where some of the inter-site links fail but others do not. In either case, the secondary site can end up in an inconsistent state.

Single-array multi-LU sequencing: maintaining the relative write order for updates across the LUs, e.g., by using a single propagation queue. *Analysis:* this works well for data whose primary copy is stored within a single array, but offers no guarantees at all otherwise. All of the single-LU propagation options discussed above apply here, too.

Multi-array propagation-cessation: since inconsistencies only occur if writes are propagated out of order, it is sufficient to stop sending updates to the secondary sites as soon as *any* write cannot be delivered. This can be achieved by having the application (or host OS) stop writing, or the disk arrays stop propagating writes, as soon as a propagation failure occurs. *Analysis:* there may be a small window of vulnerability if there are multiple outstanding independent updates while propagation-failure is detected, but this window can be bounded to roughly the duration of a long-distance timeout, or by using lock-step propagation (perhaps in conjunction with last-update rollback at the remote sites. Implementations of this approach typically require software support in the host systems (e.g., [EMC–CG2002]).

Multi-array barrier-atomic propagation: multiple disk arrays collaborate to generate *synchronization barriers*, which act to prevent updates that occur after such a barrier being applied before *all* updates that preceded the barrier have been propagated. This can be achieved via a two-phase commit protocol across the primary copy arrays (plus the hosts, if there are more than one), followed by a 2-phase commit across the secondary copies before updates are applied. *Analysis:* this is the most desirable state of affairs, but as the description suggests, it probably cannot be achieved without higher overheads and complexity than the preceding cases.

2.5 Returning acknowledgements to the host

The descriptions above considered the degree of synchronization from the point of view of the copies. Another perspective is the viewpoint of the host: just how soon is it told "we have it" on a write? Sooner means greater best-case performance, because it allows the maximum amount of concurrent I/O activity; later leaves fewer opportunities for things to go wrong after the host believes the write has been successfully recorded.

If the host issues multiple writes at a time for a single LU, the SCSI standard allows the disk array to service these in any order it finds appropriate, provided the effect is as if each write was completed in the order indicated by the request-completions sent to the host. If such writes are serviced in a different order at the primary and secondary, certain failures may make this visible.

It is helpful to separate the volatility of data from its location; in what follows, we concentrate on the volatility, remembering that these observations can be applied at both the primary and secondary copies.

Volatile immediate-report: the write is acknowledged as soon as the data is received into any storage system memory, even if this is volatile RAM. *Analysis:* Best possible performance; but data is vulnerable to a single failure, including power loss. SCSI disk drives offer this mode for data written to their cache, but it is only of use if the upper level software knows what it is doing, and it may have to explicitly sequence writes (e.g., by draining the I/O pipeline, or controlling the command queuing).

Non-volatile immediate report: the write is acknowledged as soon as the data is received into at least one non-volatile memory (NVRAM) in the storage system. *Analysis:* This is the mode commonly used by single-controller disk arrays. Data is vulnerable to loss of the single copy.

Redundant non-volatile immediate report: the write is acknowledged as soon as the data is written into failure-

tolerant non-volatile memory in the storage system (e.g., mirrored NVRAM). *Analysis:* This is the mode commonly used by high-quality dual-controller disk arrays. Data is still vulnerable to loss if the memory loses data, if the disk array suffers a complete failure, or if the site goes down.

Single write-through: the write is acknowledged only after the data has been transferred onto at least one of the long-term storage devices. *Analysis:* This is the mode used by asynchronous remote mirroring: the primary copy is the target for the write-through, the secondary, remote copy happens later. Data is vulnerable to loss of the local copy, or a site failure.

Redundant write-through: the write is acknowledged only once the data has been written through to "sufficiently many" long-term storage devices to survive the target number of concurrent covered failures. *Analysis:* redundancy can be provided by updating the local and remote copies, or by updating the primary copy and the write-behind log of pending data that needs to be sent to the secondary, remote, copies. Data is vulnerable to loss of all the copies at a site, or a site failure.

Full write-through: the write is acknowledged only once all copies have been updated. (This mode implies *lock-step* synchronization.) *Analysis:* This is the mode commonly called "synchronous". Reliability is as good as it gets; performance less so. There are circumstances when it's still the right thing to do.

2.6 Where data duplication is done

Remote mirroring requires data duplication, which can be performed in four main places, each with advantages and disadvantages. Although the relative importance of these factors changes, common concerns include: the ease of supporting heterogeneous hosts and storage devices; whether additional hardware elements are needed; whether host software needs changing or installing; additional host CPU loads; cost, performance, and scalability of the solution; and ease of supporting LU groups that span disk arrays (simplest at the host, hardest at the arrays).

At the host: typically in a device driver or logical volume manager (LVM). *Analysis:* typically gives the greatest flexibility in terms of WAN link support; allows for file-level replication, not just at the LU level; simplifies the grouping case; but imposes additional CPU load on the hosts, and may be harder to manage if the ratio of hosts or host types to storage devices is large.

In an I/O card at the host: typically by means of a "smart" host bus adapter (HBA). One example is the original COMPAQ VersaStor scheme [Widen2000]. *Analysis:* can off-load I/O processing from the host, and avoids OS-dependence of the host-based scheme. Relies on the SAN to provide in-band connectivity to inter-site links, e.g., by running IP over Fibre Channel to a gateway.

At the disk array: typically in the array controller firmware. Current implementations tend to require the remote array to be from the same manufacturer (and sometimes the same model), as the primary array. *Analysis:* probably the most commonly deployed scheme today. In the past, fewer native network-link types were supported than with host-based approaches; more recently, gateways and protocol converters have broadened the scope of support.

In the SAN fabric: either as an in-band "SAN appliance", or as an extension to a SAN fabric switch. An example of the former is the HP CASA product [HP–CASA]. *Analysis:* conceptually elegant, but may be limited by the I/O bandwidth or latency of the in-band hardware. Bounding the traffic to just the LUs in a group (in the sense of section 2.4) can be used to achieve scalability: one box needs only to handle the traffic of a single group, or a small number of them. It's also important that the appliance doesn't become a single point of failure: most appliance implementations support failover pairs for this reason. Switch-based implementations are actively being discussed in industry; how failure-tolerance will be achieved in this case is not yet clear.

2.7 Additional features

In this section, we summarize some additional techniques and possible extensions.

Multiple secondary copies. Although we have chosen to focus on the 2-copy case for ease of exposition in this paper, it's clear that there can be more than one secondary copy, and the different copies can have different techniques applied to them.

Partial copies. The descriptions above are written as if each copy is a full instance of the data. There are circumstances in which this need not be the case: what matters is that the mirroring system can provide the illusion of a full copy, even in the face of failures. Thus, an old copy plus a redo log can be used to provide the illusion of an up-to-date copy and a prior one. So can a current copy and an undo log. Similarly, just the log of recent changes can itself act as a virtual copy if there are other mechanisms for restoring the underlying state: this means that a 2.5-copy system could be constructed with the log held at an intermediate site, separate from the true secondary copy.

Bidirectional mirroring: some systems allow both ends to act as primaries for different LUs.

Active-active mirroring: two or more sites can update the same LUs. This requires both synchronous mirroring and application-level support.

system	*divergence*	*propagation order*	*where done*
Veritas Volume Replicator	none / IO-count-bounded / log-space-bounded / unbounded (bitmap)	lock-step / in-order asynchronous / in-order asynchronous / out of order	host
IBM's Peer-to-Peer Remote Copy (PPRC)	none / unbounded (full copy)	lock-step / out of order	disk array
IBM's Extended Remote Copy (XRC).	space-bounded / unbounded (full copy)	in order asynchronous / out of order	host + disk array
EMC symmetrix SRDF	*Synchronous:* none / unbounded (per-track bitmap)	lock-step / out of order	disk array
	Semi-synchronous: IO-count-bounded (<=1) / unbounded (track "bitmap")	in-order / out-of-order	disk array
	Adaptive Copy: track-count-bounded / unbounded (track "bitmap")	out of order / out-of-order	disk array
NetApps SnapMirror	unbounded (file system structure)	atomic in-order asynchronous batched, with overwrites	file server
HP XP continuous access	*synchronous:* none / unbounded (bitmap)	lock-step / out of order	disk array
	asynchronous: cache-space-bounded / unbounded (bitmap)	in-order asynchronous / out of order	disk array
HP Continuous Access Storage Appliance (CASA)	*synchronous*: none / unbounded (bitmap)	lock-step /out of order	duplexed SAN appliance
	asynchronous: disk-queue-space-bounded / unbounded (bitmap)	in order / out of order	duplexed SAN appliance
Seneca	time-bounded batches / log-space-bounded / unbounded (bitmap)	grouped, atomic in-order asynchronous batched, with overwrites / (same) / out of order	duplexed SAN appliance

Table 1: a few sample remote mirroring systems. A "/" between steps indicates the next level of fallback behavior. Additionally, in most cases, synchronous mode can be told to allow or fail (abort) application I/Os if the link goes down; these variants are not shown.

Multiple recovery points: both the batch- and log-based approaches lead themselves to a strategy of preserving prior state, rather than discarding it as soon as an update is applied. This can permit state reconstruction at multiple points in the past. Tape backups can be thought of as one extreme form of this; the S4 project at CMU, which keeps the complete write log, another [Strunk2000].

2.8 Application fail-over

We have concentrated here on the recovery and propagation behavior of the storage system. In real life, the recovery and failover behaviors of the applications are also vital, but it is beyond the scope of this paper to address these aspects, other than to observe that recovering a consistent copy of the stored data is but the first step to application recovery.

2.9 Applying the taxonomy

We found it instructive to test the taxonomy by applying it to a number of extant remote mirroring products. Table 1 provides such a sample. In addition, we expand on a few of them here to illustrate the remote mirroring design space in a little more detail.

EMC SRDF: EMC claims to have been the first to market with a "storage-based replication software application" in 1993 with SRDF, the Symmetrix Remote Data Facility [EMC2002, EMC–SRDF], which provides inter-array remote mirroring. Up to two remote copies can be maintained from one primary using the "Concurrent SRDF" feature, in synchronous mode.

EMC literature encourages the use of synchronous mode, which is a lock-step, no-divergence protocol. When the performance penalty of synchronous operation is too high, semi-synchronous mode is available, which allows one outstanding I/O per LU to be in flight asynchronously to the secondary site. Subsequent writes are stalled until it returns. Adaptive copy mode provides track-count-bounded asynchrony, with no ordering guarantees; if the count is exceeded, writes are stalled.

Multi-array propagation cessation is supported in synchronous mode, with help from EMC-supplied host software [EMC–CG2002].

If the remote site becomes disconnected, a per-track data structure keeps a note of which tracks have been modified – but not the order in which this is done (i.e., this is effectively a bitmap-like solution), so recovery from any extended link break or site outage is always performed in unordered mode, which is vulnerable to a rolling disaster. To avoid this, EMC suggests the use of the separate TimeFinder product to provide the effect of (large) batches.

IBM XRC: IBM's Extended Remote Copy (XRC) software for their MVS mainframes employs a host-based data mover to drain cached data from disk arrays' NVRAM cache, write it to an on-disk log, and transmit it to a remote site [IBM1997]. Each data mover session groups and orders the I/Os for the volumes under its care in timestamp order. If the disk array cache fills up, application I/Os are stalled, up to some threshold time, after which XRC reverts to the unbounded case, and a full volume copy will be needed.

HP XP arrays: the HP XP disk arrays and the similar Hitachi Data Systems arrays provide both synchronous updates and an in-order, space-bounded asynchronous update mode [HP–XP1024, HP–XP–CA]. The latter mode keeps writes in the mirrored NVRAM cache of the primary array; the ordering is implemented by maintaining a timestamp or sequence number on dirty blocks in the cache, and applying the writes in the same order at the secondary. Multiple LUs in one array can be grouped into a consistency group, which is the unit of I/O ordering. The amount of cache allocated to the write buffer is sized dynamically; when it gets above a high water mark, foreground application writes will be deliberately slowed down in increments, eventually to virtual lock-step. As we will show later, the array's cache is likely to provide adequate buffering in the absence of a link or remote site failure without any slowdown. As a last resort, the array reverts to bitmap-based unbounded divergence.

HP Continuous Access Storage Appliance (CASA): a SAN mirroring appliance [HP–CASA]. In asynchronous mode, it provides in-order delivery until a disk-based queue fills up, and then resorts to an unbounded bitmap mode to track updates during any remote site or link failure.

Veritas Volume Replicator (VVR) [Veritas2002]: a host-based scheme that can support multiple remote copies. It can operate in synchronous mode, or with per-host I/O-count-bounded, multi-LU, in-order asynchrony, using a circular disk-based "storage replication log (SRL)". If the remote site becomes unreachable in synchronous mode, then writes can be refused, or VVR can drop into "soft" asynchronous mode until the link comes back up. If the SRL fills up, writes can be stalled until space is available, refused, or VVR can drop into an unbounded mode that uses a bitmap to record updated blocks. If a count-based bound for asynchronous I/Os is reached, writes can be refused or stalled until a low water mark is reached. VVR does no overwrite absorption in normal operation, and nor does it perform atomic updates at the secondary. Writes are acknowledged from the secondary copy as soon as they are received in the secondary host (a special case of the volatile immediate report of section 2.5).

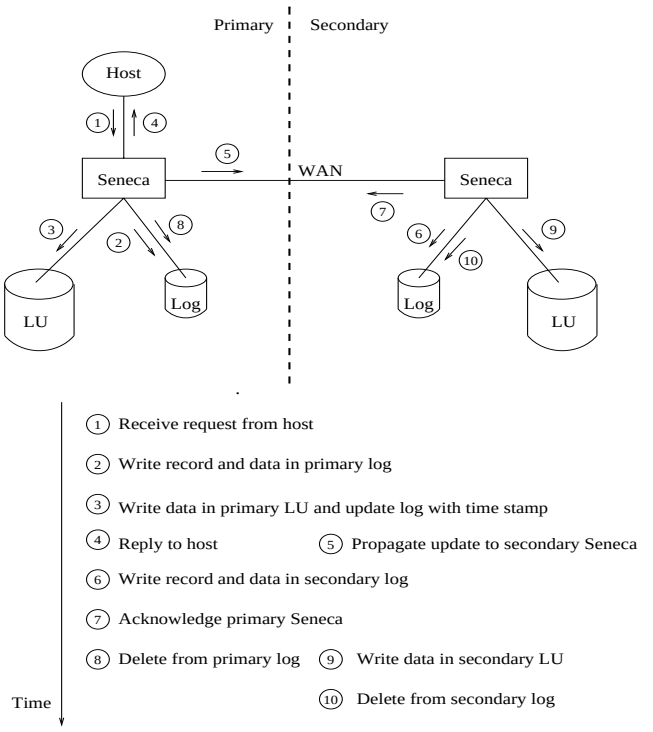


Figure 3: the general steps in the Seneca mirroring protocol.

Network Appliance SnapMirror: this product uses asynchronous, write-coalescing batched file system updates that are applied atomically at a remote file server [Patterson2002]. The WAFL file system is used to keep track of the blocks that have been updated, including which blocks are still in use and worth propagating. The WAFL file system always operates in a no-overwrite mode, so it readily supports applying a coordinated set of updates atomically.

3 The Seneca protocol[1]

For Seneca, we set out to design an asynchronous mirroring system that could exploit a relatively low-speed wide-area network link by coalescing overwrites safely – i.e., in an order-preserving manner. Seneca's goal is to make the data available and keep each copy consistent despite disk array

[1] "Seneca, comparing his degenerate times with those of the heroic Scipio, who bathed in austerely simple surroundings, complained: 'But who in these days could bear to bathe in such a fashion? We think ourselves poor and mean if our walls are not resplendent with large and costly mirrors …'." [de la Croix1975, p225].

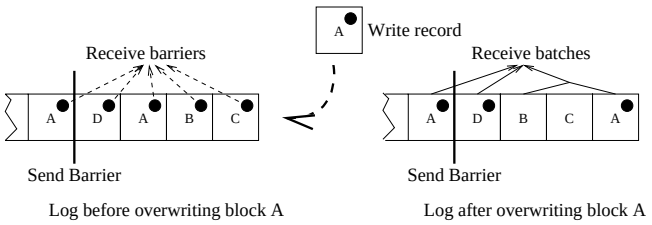


Figure 4: receive batches.

failures, Seneca box failures, network failures, and both temporary and permanent site outages.

To simplify the exposition, we present the Seneca protocol as if it were running between a dedicated pair of remote-mirroring SAN appliances (Figure 1), although the protocol could readily be implemented in the disk arrays, in the hosts, or in host I/O bus adapters.

An overview of the protocol is shown in Figure 3. In normal operation (i.e., while the remote site is reachable), a write at the primary causes a *write record* to be written synchronously in a *primary log*, and the data can then be written to the primary copy (LU). The primary log can be implemented in a disk inside a Seneca box, or, better, in a disk array in the local SAN—preferably one equipped with an NVRAM write buffer, and disjoint from the one holding the primary copy.

Seneca implements the overwrite log with an atomic update protocol. It can use either the single or redundant write-through schemes to report completion back to the host; and it can be integrated into a multi-LU propagation scheme.

3.1 Log barriers

Update records and data blocks in the primary log are propagated to the secondary Seneca in the same order as they were written to the primary LU. This transmission occurs in parallel with continuing operations at the primary; we assume in-order transport across the WAN. The propagated blocks are appended to the log at the secondary Seneca. This *secondary log* is divided into *receive batches*, which are committed to the secondary LU atomically. Receive batches are bounded by inserting *send barriers* and expanded by removing *receive barriers*.

A *send barrier* is one that Seneca deliberately and periodically inserts into the primary log, following the last block that has been written to the primary LU. Overwrites to blocks following the last send barrier are allowed, which saves both network bandwidth and log space. Overwrites to blocks preceding any send barriers, however, are prohibited. Send barriers may bound the divergence between the primary and secondary Senecas in terms of the elapsed time, the number of transactions, the amount of data, or any other metric. Their frequency can be adjusted manually or automatically, as described in section 2.3.

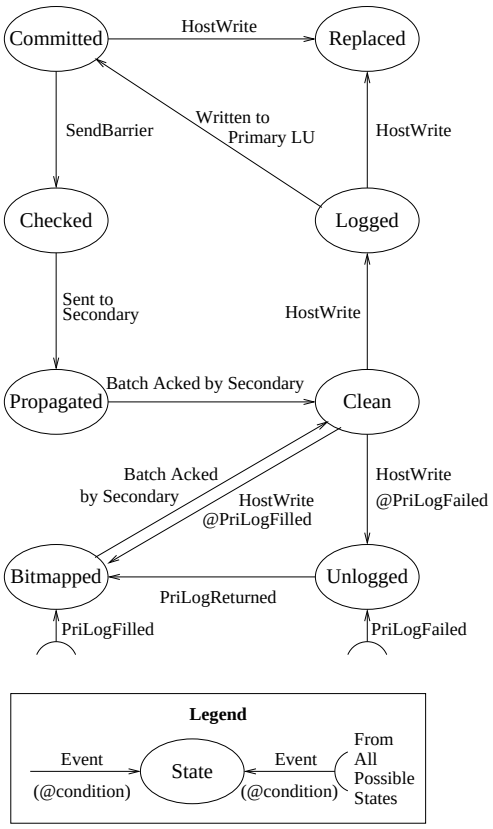


Figure 5: data block states in the primary Seneca.

A *receive barrier* is initially associated with each write; that is, each data block can by default be committed at the secondary Seneca by itself. Receive barriers bound the sets of blocks that have to be applied atomically as a unit. When a block is overwritten before it is propagated, the earlier write record for that block is removed from the log, as are any receive barriers for blocks written between the old copy and the new write (i.e., the end of the log). This merges those blocks into the same receive batch.

For an example, consider Figure 4. The second write to data block A overwrites the earlier version, and removes the receive barriers associated with blocks B and C that were written between the two writes to A. The consequence of the overwrite is that the three blocks A, B and C can no longer be transferred to or committed in the secondary Seneca in the same order as they were written—instead, either all of them need to be committed at the secondary copy, or none.

3.2 Block state transitions

Figure 5 shows the state transitions of a data block and its write record in the primary log.

A data block is in the clean state if it has not been written since the last time it was committed at the secondary Seneca. A write to a clean block normally adds a *write record* and a copy of the newly-written block to the primary log and

changes the state of the block to logged. After the block is written to the primary LU, its state changes to committed. A write to a logged or committed block overwrites the copy in the primary LU, and appends a new write record (with state logged) and a copy of the data to the primary log, effectively replacing the old record and data.

A committed block changes to the checked state after a new send barrier is inserted (after it) into the log. A checked block changes to the propagated state after its record and data are sent to the secondary Seneca. A write to a checked or propagated block overwrites the copy in the primary LU, and appends a new write record and a copy of the new data to the primary log, which do not replace the old record and data. An acknowledgement from the secondary Seneca for a batch containing the propagated block changes the block back to the clean state.

When the primary log space is filled, all logged, committed, checked and propagated blocks change to the bitmapped state, which will be explained in Section 3.3.

At the secondary Seneca, the newly-received records and data blocks are first appended to a *secondary log* there, typically on disk, although it could be in NVRAM. When a receive barrier is stored in the secondary log, the secondary (1) sends an acknowledgement for the receive batch back to the primary Seneca, (2) marks the batch as "acknowledged" and (3) triggers the batch's commitment at the secondary LU. The commitment copies the data blocks from the secondary log to the secondary LU, deletes the log records, and frees the log space.

Acknowledgements sent to the primary Seneca are used to garbage-collect the primary log: an acknowledged receive batch allows all the records and data blocks in that batch to be deleted from the primary log.

Both the primary and the secondary Seneca can fail during the propagation process. When it recovers, a Seneca instance examines the kind of log it had to determine whether it was the primary or secondary Seneca. If it was a secondary Seneca when it failed, it will re-apply its secondary log up to and including the last acknowledged receive batch, and carry on as a secondary. The situation for a returning Seneca with a primary log is more complicated, which will be discussed in section 3.3.4.

3.3 Seneca state machines

The Seneca protocol is defined in terms of a pair of state machines, one for each of the primary and secondary Seneca modes. Figure 6 shows the state transitions of a Seneca instance in response to failure and recovery events.

When acting as a primary Seneca, the possible failure events are PriLogFilled (primary log filled), PriLogFailed (primary log disk failed), PriFailed (primary Seneca or LU failed), and SecDisconnected (any failure that leaves the secondary Seneca inaccessible to the primary one, including secondary log disk failure, secondary Seneca LU failure and/or network outage).

The possible recovery events include PriLogReturned (primary log disk is repaired), SecRepaired (secondary returns and contains all the data stored in its log and LU before it crashed) and SecReplaced (secondary returns with empty storage).

Acting as a secondary Seneca, the possible failure events include SecLogFilled (secondary log filled), SecFailed and PriFailed. The possible recovery events include SecRepaired and SecReplaced.

We describe a few interesting corner cases in the Seneca state transitions below.

3.3.1 Failover and fallback

In case of a secondary failure, the primary Seneca continues to perform in the Standalone state, and brings the secondary site back up to date when it returns.

If the primary Seneca fails, the secondary Seneca becomes the new primary. We call this *failover*. When a Seneca returns, it always returns as a secondary Seneca. Even if it was the local Seneca, it needs some preparation, such as cleaning up its log, before it is ready to serve as the primary. When both Senecas are ready, i.e. free of outstanding updates, the local Seneca becomes the primary again. We call this *fallback*.

We assume that some entity outside Seneca will make the decision of whether and when to failover or fallback, and notify Seneca accordingly, because this decision is influenced by many factors outside the storage system, including the state of networks, sites, and applications, as well as the judgement of human operators. It may also involve operations at application level to complete the failover.

Seneca's failover mode starts when the surviving secondary changes to the Failover state. In the Failover state, the secondary prepares to be the primary by first committing the data in its secondary log to its LU, to bring the LU as up to date as possible. Write requests will be put on hold until Seneca leaves this state, e.g. changes to the Standalone state.

When it is repaired, the secondary Seneca first sends a request to the primary for updates, and then starts to operate in the Normal state. However, if it returns with empty storage, Seneca starts in the DirectUpdate state because the complete snapshot of the primary LU will be propagated and there almost certainly won't be enough log space for the complete snapshot. After the entire snapshot is committed in the secondary LU, the secondary Seneca changes to the Normal state.

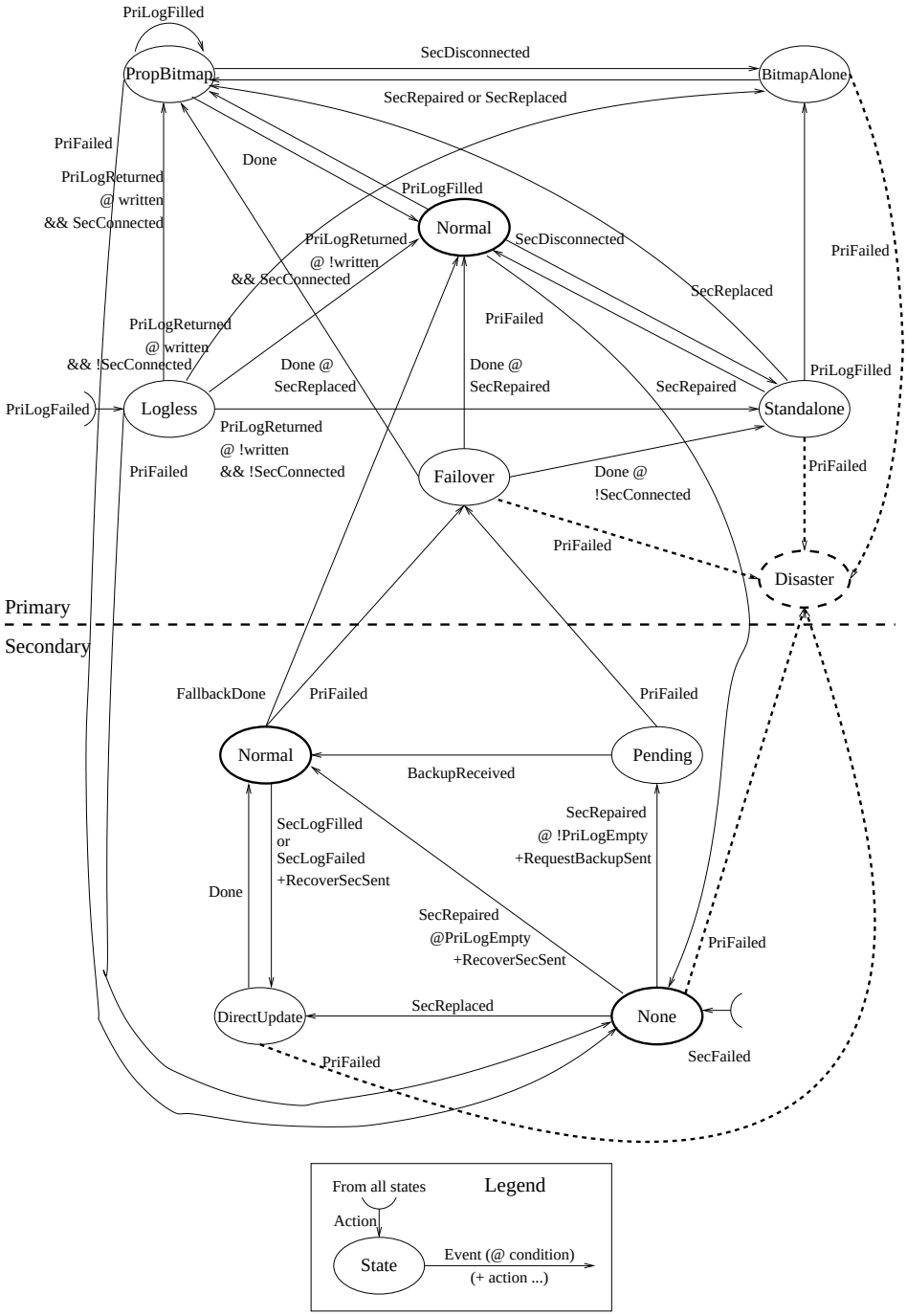


Figure 6: the Seneca state machine. “Primary” and “Secondary” refer to the mode in which a Seneca instance operates.

3.3.2 Log failure

In the Normal state, when the primary log space is filled (e.g., because the secondary Seneca is down for a long enough time, or the network connection to the secondary is too slow), or the log fails, the primary Seneca changes to the PropBitmap state, and merely marks new updates in its bitmap.

In the PropBitmap state, the primary Seneca propagates the blocks in the bitmap (or a complete snapshot of the primary LU if necessary) to the secondary as soon as it can, and asks the secondary to commit all the changes to its LU atomically if possible. However, the secondary may not be able to do so if it is in the DirectUpdate state. As a consequence, the data in the secondary LU could be inconsistent with that in the primary LU until all the changes are committed.

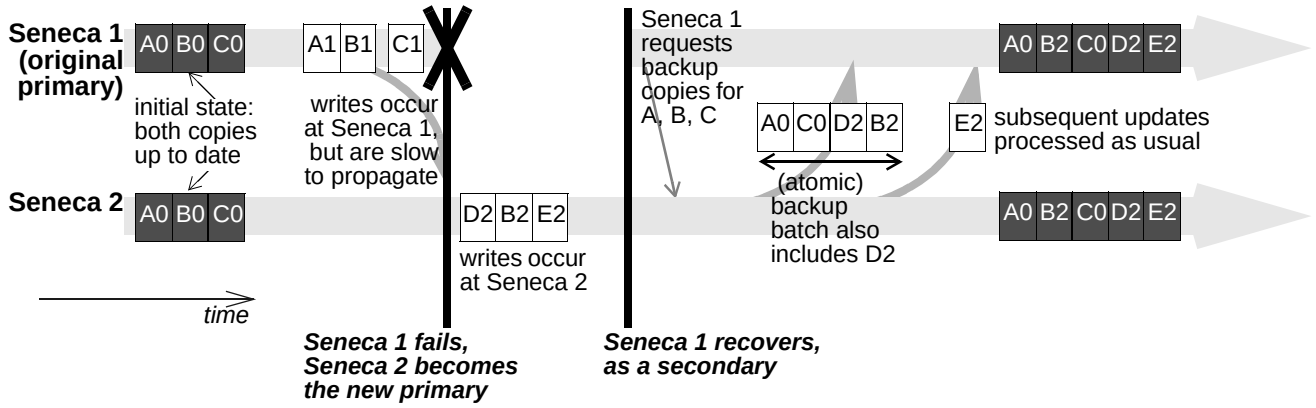


Figure 7: update and backup-copy propagation during failover and recovery. Shaded blocks are in the local LU at each Seneca, unshaded blocks also reside in the local log.

When the primary log fails in any state, the primary Seneca changes to the Logless state, in which it does not record any updates and does not propagate any updates to the secondary.

In the Normal state, when the secondary log space is filled or fails, the secondary Seneca changes to the DirectUpdate state, in which it writes any data from the primary directly to the LU, bypassing the log. This may mean that the data in the secondary LU is inconsistent with the primary copy in the DirectUpdate state—but the only alternative is to allow it to get still further out of date.

3.3.3 Disaster

When both Senecas fail simultaneously, the whole system is out of service, i.e. it changes to the Disaster state. If the primary fails while the secondary is in the DirectUpdate state, the whole system is also out of service because the surviving secondary is not able to serve consistent data.

3.3.4 A failover dilemma

When a primary Seneca fails, applications can fail over onto the secondary Seneca, which then becomes the new primary. Data left in the old primary log will be considered lost, and new data can be written to the new primary Seneca. When the old primary Seneca returns, the data in its LU and log could be inconsistent with that in the new primary Seneca, i.e. both Senecas could have a piece of data that the other does not. The following options exist:

1. Prohibit failover. The entire storage will be unavailable until the old primary Seneca returns, but inconsistency will not occur.
2. Live with inconsistency and minimize data loss. The old primary Seneca will keep the data in its log, and people or applications must resolve the conflicts between the two mirror sites.
3. Live with data loss and eliminate inconsistency. The old primary Seneca will try to undo the data in its log by requesting the current copy from the new primary.

Both options 1 and 2 require intervention at the application level, and so are outside the scope of Seneca's ability to recover. In practice, option 2 is usually the one that is used, but we focus here on option 3, which requires undoing changes in the old primary's log. Figure 7 shows an example.

In the old primary log, some blocks might have been committed to the local LU while others have not. The old primary Seneca asks the new primary for its current copy of the committed blocks, which we call the *backup* of those blocks. The backup of all committed blocks is transmitted to the old primary as a single *backup batch*, and will be written to the old primary LU in a single, atomic action.

In a more complicated case, some committed blocks might have been written in the new primary Seneca before the old returns (e.g., B2 in Figure 7). In this case, the backup batch will be expanded to include all other blocks that were written before the committed blocks in the new primary (e.g. D2 in Figure 7). The reason is similar to that for atomic receive batches (Section 3.1). Once the committed updates at the old primary are successfully undone, the old primary log is simply discarded.

In an extreme case, the new primary Seneca might fail before it sends the backup to the old one. In this case, the old primary Seneca will keep its log and try to commit the logged blocks in it instead. As a consequence, data considered lost in the failover mode (e.g., A1, B1 and C1 in Figure 7) can come back while data written in the failover mode (e.g., D2, B2 and E2 in Figure 7) can disappear after the new primary fails!

3.4 Seneca shadowing

Seneca itself needs to be fault-tolerant; therefore, a local Seneca instance is implemented as a *pair* of Seneca boxes.

The simplest way to achieve this is to have one Seneca box be *active*, while the other acts as a *shadow*, to which the active Seneca can fail over. Changes to the active Seneca's memory are propagated to the shadow before their write operation returns (e.g., in the style of Harp [Liskov1991]). Active-active implementations are also possible.

Ideally, both Seneca boxes will have access to a common log on the SAN, and the only extra latency is the transferring of a short message to the shadowing Seneca across an interconnecting LAN. Therefore, the shadowing Seneca will be a hot standby, i.e., it will have exactly the same state as the primary one when the fail-over occurs. The log disk can be made fault-tolerant by local redundancy schemes, such as RAID5, or mirroring.

3.5 Summary

In this section we presented a remote mirroring protocol in some detail, including the corner cases that make such protocols so challenging in practice.

4 Verifying Seneca correctness

Given the complexity of the corner-cases in a protocol of this form, we felt that it was important to apply some kind of assurance testing to it beyond just prototype and test. Sample approaches include theorem proving, model checking, and simulation. We quickly learned that the Seneca protocols were too complicated to describe at a useful level of detail in any of the languages used for theorem proving and model checking, and cast around for an alternative.

We started from I/O automata [Lynch1989], which model components in asynchronous concurrent systems as labeled transition systems. While the original I/O automata method uses a special purpose language to allow an exhaustive search of the automata state space, our approach was simulation: we built an event-driven simulator that generated events by constrained random walks in the state space, executed the automata, and checked the correctness of the results against our expectations of correct behavior and event sequences. This had the added benefit of allowing the protocol to be implemented in the same language as a real deployment (in our case, C).

It is not our intention here to argue that this approach was the best possible one, but instead to report on our experiences with what turned out to be a fruitful tool.

We wrote automata for each Seneca box, log, LU, WAN and write record, and modelled the Seneca protocol responding to external events. State transitions in the automata take place in response to external events such as write request, log disk failure, Seneca or LU failure and WAN outage. State transitions also take place in response to internal events between the machines such as log space exhaustion, update propagation and batch commit.

We used the model to increase our confidence in:

1. *Coverage:* that Seneca is in a valid state after any sequence of events.
2. *Safety:* that the mirror copies are consistent in normal states, and the sequence of updates at the secondary Seneca is always a prefix of the sequence at the primary.
3. *Liveness:* that data will eventually be written in both mirrors unless a disaster happens.

To check liveness, we stopped the generation of external events and let the automata run until no more internal events were generated. We used assertions inserted in various locations of the model for correctness checking, rather than formal reasoning. We found this approach to be an efficient way of checking a complicated protocol like Seneca, and gaining confidence in it.

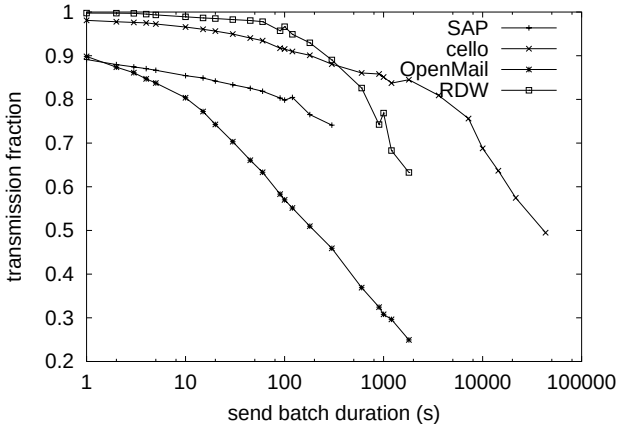
The checking was done incrementally. The simulator injected a random sequence of external events into the Seneca model until an assertion failed. We then looked for the bug by examining the simulation traces. On average, it took less than half an hour to discover and fix a bug. Typically, the model would survive a longer sequence of events after each bug was fixed. We biased the event stream very strongly towards failure events, in order to test the recovery code: typically 15 writes for every external (failure or recovery) event.

Over 131 runs with distinct random seeds, the average count of failure injections before a protocol error was detected was 16435. If we assume that the Seneca's handling of write events is correct, and external failures (such as the failure of a mirrored Seneca or log device) occur at the (relatively high) rate of once a quarter, this corresponds to an estimated mean time between failures (MTBF) of 4100 years for the Seneca protocol proper. Obviously, this is a statistical estimate, not a guarantee, and does not take into account implementation or operator errors—but we still feel that it builds more confidence than merely asserting the protocol's likely correctness.

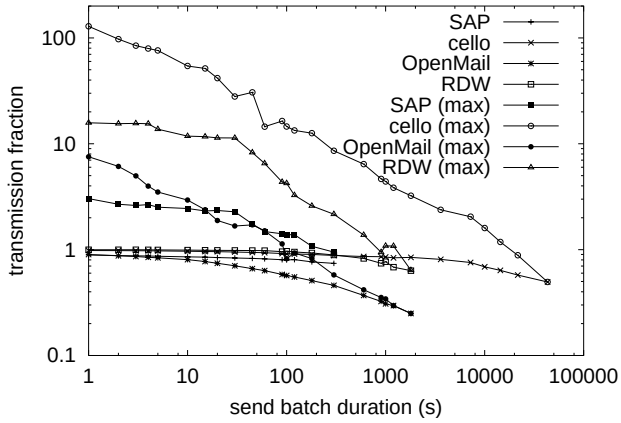
Most of the bugs we found so far were transcription errors in coding the state-transition diagrams we had developed, which would have been too detailed to be covered by a theorem prover or a model checker. In one example, a backup batch (section 3.3.4) included new updates that had already been propagated to and queued in the secondary Seneca, so the queued updates were committed twice, resulting in out-of-order writes to the secondary LU. This kind of error would be hard to detect with traditional testing—especially if failures were induced manually. The last bug we had found at the time of writing was only detected after 1.77M writes, 75.9k failure events and 22.4k recovery events were injected into Seneca, which in turn provoked Seneca to generate an additional 6.60M internal events.

workload	*length*	*date*	*I/O count*	*write count*	*% writes*	*total data written*	*mean write rate*	*peak 1 second write rate*
cello2002	24 hours	2002/09/18	6760626	5250126	77.7%	67.40 GB	0.78 MB/s	100.30 MB/s
SAP	15 mins	2002/01/31	4986071	150339	3.0%	1.75 GB	1.95 MB/s	6.75 MB/s
RDW	1.4 hours	2000/02/11	1797210	60758	3.4%	1.70 GB	0.34 MB/s	5.33 MB/s
OpenMail	1 hour	1999/12/17	1287941	931979	72.4%	6.13 GB	1.70 MB/s	15.80 MB/s

Table 2: traced system workload summary



(**a**) Mean transmission fraction (averaged across all the batches), scaled to the mean write rate.



(**b**) Largest-batch transmission fraction, scaled to the *mean* write rate, as well as the mean transmission fraction, for scale. Note the log scale on the Y-axis.

Figure 8: fraction of data transmitted across the WAN, as a function of the batch duration. The left-most point corresponds to no overwrites (for which batch duration is irrelevant); the others are for the indicated batch duration. Both graphs are scaled so that 1.0 corresponds to the mean write rate (see Table 2 for its value).

5 Performance analysis

One of the key questions for Seneca is the amount of network bandwidth that can be saved by delaying I/Os (asynchrony) and coalescing (over-writes). To assess how this will work in practice, we examined a number of real-world I/O traces, summarized in Table 2. The traces were gathered from HP-UX systems, using techniques similar to those used in [Ruemmler1993].

- cello2002: an 8-processor HP 9000 N4000 timesharing system for a small group of researchers with 16 GB of RAM, an HP XP512 disk array (total allocated storage was 1.44 TB), running HP-UX 11.00; the trace data is from Wed Sept. 18th, 2002. This is a successor to the 1992 cello system described in [Ruemmler1993].
- SAP: an SAP installation from a utility company, running on an HP V2500 that was using SAP ISUCCS 4.5B on top of a 4 TB Oracle database, being accessed by more than 3000 users retrieving customer's utility bills for updating and reviewing. There were also some batch jobs running in the background. The trace was taken in the afternoon of Thursday Jan. 31, 2002. The storage system was an HP XP512 disk array with 160 73 GB disks in RAID 1/0 mode, a 16 GB cache, and running remote mirroring to a second, remote XP512 via eight ESCON links.
- RDW: a retail data warehouse system, containing 500 GB of shopping basket information, executing on an HP V2250 running HP-UX 11.00; the storage system was an EMC Symmetrix array. The trace was made on Tuesday Feb. 22, 2000.
- OpenMail: one of 6 HP9000 K580 servers, each of which had 6 CPUs and 3.75 GB of memory and was connected to an EMC Symmetrix 3700 disk array providing 640 GB of storage. The system was running a production OpenMail email workload on HP-UX 10.20 with 4500 users, 1400 of whom were active during the traced period. The trace was taken on Thursday Jan. 20, 2000.

We begin by examining how much the overwrite rate reduces the WAN traffic: quite respectable reductions in total WAN traffic are available: e.g., 5–40% for a batch size of 30 seconds (Figure 8a), and significantly greater reductions for the largest batches (Figure 8b).

Figure 9 confirms that the distribution of batch sizes is far from uniform, which is why the rate-smoothing performed

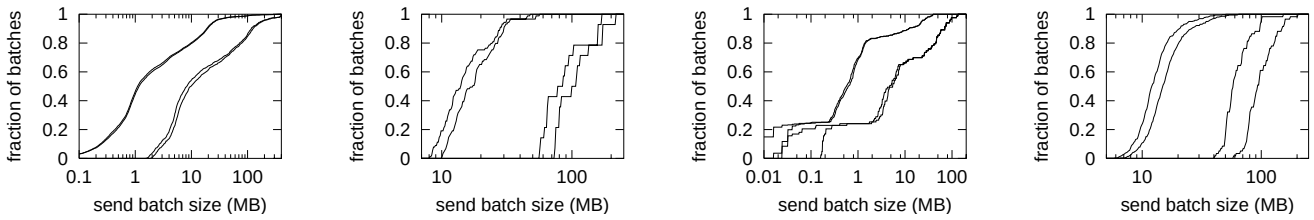


Figure 9: CDFs of measured batch sizes for 10 and 60 second batch durations, with and without overwrites, for the traced systems. The pair of lines to the left are the 10-second case, the leftmost line within each pair is the no-write-coalescing case.

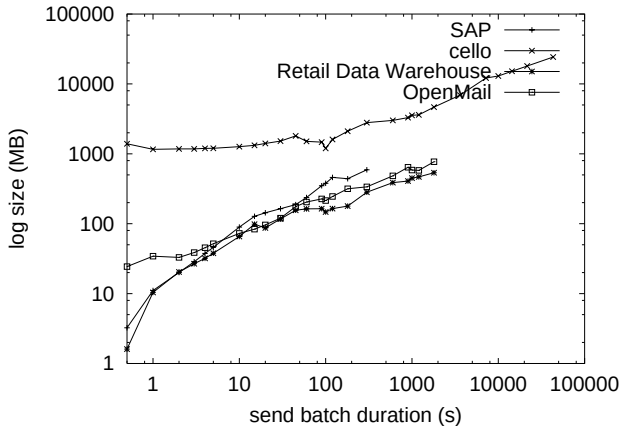


Figure 10: worst-case log size as a function of batch size. The left-most values denote an adaptive-duration batch: the batch end (send barrier) was inserted as soon as the transmission of the previous batch finished.

by asynchronous I/Os is so important. The long-term mean write rate for each of our workloads could be handled by a single 45 Mb/s T3 line, even with no overwrite absorption. But the peak rates require more: the 95th percentile of the 10 second-average I/O rate (Figure 9d), scaled up to cover all 6 OpenMail servers, would require three T3 lines at 85% utilization with write coalescing, and four lines at 93% utilization without.

We determine the *worst-case* log size for each trace, using the 100% T3 link utilization assumption. Figure 10 shows that the log size required to handle write propagation delays is small when the link is up – usually much less than 1 GB. (Cello is an anomaly: it was processing large I/O traces at the time, and its load is a file system, whereas the other systems are databases.) This means that the size of the log is determined almost entirely by the link outage time for which in-order delivery is desired: at these average rates, even a 100 GB log will cover an outage for 14–81 hours.

In some cases, the cost for a WAN link is a function of the data sent over it. To explore the effects of this, we extrapolated the OpenMail workload for all six servers, and applied the cost functions from [SC2002a] to the WAN traffic that resulted. Using no overwrites, the cost would have been $40.1k/month; allowing write coalescing reduced the cost to $31.3k/month with a 1 minute batch duration, or $12.7k/month with a 5 minute batch duration.

6 Related work

Our goal here is to discuss the genesis/predecessors for the Seneca work itself, rather than existing mirroring products, which were discussed above.

In an analysis of the failure statistics of the Tandem fault-tolerant system [Gray1986a], Jim Gray suggested that remote replication, if one could afford it, protected against 75% of all failures. Partial replication, in the style of RADD [Stonebraker1989], information dispersal [Rabin1989] or Myriad [Chang2002], is probably inappropriate for the applications considered here because of their I/O latencies.

The Jasmin reliable disk server [Uppaluru1987] created a watch dog to copy disk partitions from the active server to the semi-active (once-failed) server, and maintained a water mark to indicate which blocks had been copied and which had not.

Mime [Chao1992], a shadow-writing storage system, avoided synchronous metadata updates using a log and barriers.

The SnapMirror paper [Patterson2002] contains some similar observations to ours, but its focus was on execution performance, rather than the details of the protocol or its correctness; long batch lengths—tens of minutes to hours, rather than seconds; and the support needed in the WAFL no-overwrite file-system for block coalescing, rather than how to minimize the size of the atomic update groups, or the correctness of the resulting protocol.

7 Conclusion

In this paper, we explored the complications and nuances of the remote mirroring problem and the design space of solutions to it, by providing a taxonomy of both. We also described—in some detail—a protocol for asynchronous write propagation that allows safe overwrites, and told of our experiences in validating it's correctness and assessing its performance.

Our studies of real-world trace data suggest that long-term average write bandwidth may be quite low, even if peak I/O rates are high. These results show that asynchronous remote mirroring protocols can deliver significantly lower WAN link traffic by smoothing out bursty write traffic, and write coalescing reduces the WAN traffic still further – especially in the bursty periods. Doing so correctly and safely requires careful attention to avoiding many possible failure modes and corner cases. We believe Seneca provides such a protocol.

Acknowledgements

Thanks to David Black and Bob Cochran for helping us understand the intricacies of the remote mirroring space, and to Marvin Theimer, our USENIX shepherd.

References

[Chang2002] F. Chang, M. Ji, S. A. Leung, J. MacCormick, S. E. Perl, and L. Zhang. *Myriad: cost-effective disaster tolerance*. In: *Proc. FAST* (Monterey, CA) pages 103–116, Jan. 2002, USENIX.

[Chao1992] C. Chao, R. English, D. Jacobson, A. Stepanov and J. Wilkes. *Mime: a high performance parallel storage device with strong recovery guarantees*. HP Laboratories technical report HPL–CSP–92–9rev1 (March 1992, revised November 1992). Available from http://www.hpl.hp.com/SSP/papers.

[de la Croix1975] H. de la Croix and R. G. Tansey. *Gardner's Art Through the Ages*, 6th edition, 1975, Harcourt Grace Jovanovich: New York.

[EagleRock2001] *Online survey results: 2001 cost of downtime*. Eagle Rock Alliance Ltd.http://contingencyplanningresearch.com/2001%20Survey.pdf, accessed Apr. 2002.

[EMC2002] *Timeline*. EMC Corporation. http://www.emc.com/ip/timeline.html, accessed Apr. 2002.

[EMC–CG2002] *EMC Enterprise SRDF consistency group: description and usage validation*. Engineering white paper, Feb. 2002. EMC Corporation.

[EMC–SRDF] *Symmetrix Remote Data Facility product description guide*. June 2000. EMC Corporation.

[EMC–TimeFinder] *EMC TimeFinder product description guide*. Dec. 1998. EMC Corporation.

[HP–CASA] *hp OpenView continuous access storage appliance*. SAN white paper, Nov. 2002. Hewlett-Packard Company.

[HP–XP1024] *hp disk array xp1024*. Product brief 5981–0405EN. Apr. 2002. Hewlett-Packard Company.

[HP–XP–BC] *hp surestore business copy xp*. Product brief 5980–5097EN, 2001. Hewlett-Packard Company.

[HP–XP–CA] *hp continuous access xp*. Product brief 5980–8608EN, 2001. Hewlett-Packard Company.

[IBM1997] *DFSMS/MVS Version 1 Remote Copy Administrator's Guide and Reference*. SC35-0169-03, 4th edition, Dec. 1997. IBM Corporation.

[Kondoff1988] A. Kondoff. *The MPE XL data management system exploiting the HP Precision Architecture for HP's next generation commercial computer system*. Digest of papers, *Spring COMPCON'88* (San Francisco, CA) pages 152-155, Feb.–March 1988. IEEE.

[Liskov1991] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, L. Shrira, and M. Williams. *Replication in the Harp file system*. In *Proc. 13th SOSP*, published as *Operating Systems Review* **25**(5):226–238, Oct. 1991. ACM.

[Lynch1989] N. Lynch and M. Tuttle. *An introduction to input/output automata*. CWI-Quarterly **2**(3):219-246, Sep. 1989. Centrum voor Wiskunde en Informatica, Amsterdam. Also, Technical Memo MIT/LCS/TM-373, Laboratory for Computer Science, MIT.

[Patterson2002] H. Patterson, S. Manley, M. Federwisch, D. Hitz, S. Kleiman, and S. Owara. *SnapMirror: file-system-based asynchronous mirroring for disaster recovery*. In *Proc. FAST* (Monterey, CA) pages 117–129 Jan. 2002. USENIX.

[Rabin1989] M. O. Rabin. *Efficient dispersal of information for security, load balancing, and fault tolerance*. J. ACM **36**(2):335-348, Apr. 1989.

[Ruemmler1993] C. Ruemmler and J. Wilkes. *UNIX disk access patterns*. In *Proc. Winter USENIX*, pages 405–420, Jan. 1993.

[Savage1996] S. Savage and J. Wilkes. *AFRAID—A Frequently Redundant Array of Independent Disks*. In *Proc. Winter USENIX*, (San Diego, CA) pages 27–39, Jan. 1996. USENIX.

[SBC2002] *FasTrak Asynchronous Transfer Mode (ATM) Cell Relay Service*. SBC Pacific Bell. http://www02.sbc.com/Products_Services/Business/ProdInfo_1/1,,130--1-1-0,00.html, accessed Apr. 2002.

[SC2002a] *Internet Services: E-RATE master contract number SRC26115*. Sprint Communications Company. Available from South Carolina Division of the State Chief Information Office, http://www.state.sc.us/oir/rates/docs/sprint-internet-rates.htm, accessed Apr. 2002.

[SC2002b] *SCNET ATM POP to POP pricing*. Bell South. http://www.state.sc.us/oir/rates/bellsouth/pdf/SCNET_ATM_POP_TO_POP_RATES.pdf, accessed Apr. 2002.

[Stonebraker1989] M. Stonebraker, *Distributed RAID – a new multiple copy algorithm*, Technical report UCB/ERL M89/56, Electronics Research Laboratory, University of California at Berkeley, May 1989.

[Strunk2000] J. D. Strunk, G. R. Goodson, M. L. Scheinholtz, C. A. N. Soules and G. R. Ganger. *Self-securing storage: protecting data in compromised systems*. In *Proc. of the 4th OSDI*, pages 165–180. IEEE.

[Uppaluru1987] P. Uppaluru, W. K. Wilkinson, and H. Lee. *Reliable servers in the JASMIN distributed system*. In *Proc. 7th ICDCS*, pages 105–112, Sept. 1987. IEEE.

[Veritas2002] *VERITAS Volume Replicator 3.5: Administrator's Guide (Solaris)*. June 2002. Veritas Software Corp.

[Widen2000] S. Widen. *Compaq VersaStor technology:the road to an open SAN?* IDCFlash #22488, June 2000. International Data Corporation. Available at http://www.compaq.com/products/storageworks/newsandevents/analyst/22488.html, accessed Apr 2002.

[Witty2001] R. Witty, D. Scott. *Disaster recovery plans and systems are essential*. Gartner FirstTake: FT-14-5021, 12 Sep. 2001. Gartner Research.