

Demystifying Perforce Backup and Near-Real-Time Replication

Richard E. Baum

**Senior Technical Support Engineer
Perforce Software, Inc.**

Overview

Perforce is used as a storehouse for a company's most valuable assets, its intellectual property. Too often, companies are neglectful when it comes to ensuring that this information is adequately backed up in the event of a problem. Implementing a backup strategy for Perforce can seem like a daunting task, but it really is not. There are a number of strategies available, along with ample tools, to suit the needs of any size organisation. A little bit of knowledge goes a long way and that certainly applies to this topic. Knowing about your organisation's requirements and the available tools will help you to quickly and easily put together a sound backup and recovery strategy.

This document will approach this topic by breaking it down into several logical pieces. They are as follows:

- What needs to be backed up
- Built-in Perforce functionality to accomplish this
- Backup scheduling
- What to do when a simple strategy will not work

What needs to be backed up

On the server, Perforce stores its back end data in a directory that is called the P4ROOT. This data is broken into two parts. The first is the versioned file tree, which consists of the individual file revisions that have been submitted to the repository. The second is the metadata, which contains information about the versioned file tree, such as who submitted a file, and when and from where it was submitted.

The versioned file tree consists of both binary and text files. These are only written to while a file revision is being added to or edited. The rest of the time it is only read from.

The metadata, or db.* files, on the other hand, consists of binary database files. These are read from and written to whenever users make requests of the Perforce server.

While both sets of files need to be backed up, care must be taken to not have backup, virus scanning, or other non-Perforce Server processes access the Perforce metadata files while the Perforce Server is running. Failure to follow this advice can lead to server stalls, server shutdown, or corruption within the db.* files. This might necessitate restoration from a prior backup.

The functions within the Perforce Server that support backup and recovery operate only on the db.* files. You will need to use operating system or third-party tools to back up the versioned file tree as part of your overall backup strategy. This document focuses mainly on the tools and procedures needed to handle proper backup of the Perforce db.* files.

Built-in backup/restore functionality

Checkpoints and journals

Perforce has built-in commands for making backups of its db.* files. These commands result in two kinds of files being created: checkpoints and journals. Checkpoints contain a snapshot of the contents of db.* files at a known point in time. Journals contain the transactional changes of the db.* files between checkpoints. Both checkpoints and journals are numbered. Both are plain ASCII text files. They are of the same general file format.

Perforce backup commands

When taking a checkpoint, the server locks the db.* files, dumps them to an ASCII text file, numbers the existing journal file, and starts a new journal file. It does not touch the versioned file tree. A checkpoint is the most basic backup operation available. It can be initiated from the server machine with the command, “p4d -jc”. From the Perforce client side, the command “p4 admin checkpoint” will perform the same operation.

There are several commands and arguments that allow more complex operations. (A command line flag “-z” tells Perforce to use compression.) The commands are as follows:

- p4d -jc [-z] – Take a checkpoint, rotate the journal
- p4 admin checkpoint [-z] – Client-side checkpoint/journal
- p4d -jj [-z] – Rotate the journal
- p4d -jr [-z] *filename* – Restore the journal file *filename*
- p4d -jd [-z] *filename* – Checkpoint to *filename* without incrementing the checkpoint/journal counter
- p4d -c *command* – Lock the database files, and run *command*.

In order to ensure a consistent backup, make sure that only the Perforce server process, p4d, accesses the db.* files that contain your depot metadata. System-level backups that attempt to copy these files can cause problems. If Perforce attempts to access them while the backup process is doing so, it could cause the server to shut down improperly if it could not obtain exclusive locks on them. The same is also true of virus scanners.

If you were able to copy the files without causing problems for Perforce, the resultant backup would almost certainly be of little use in the event of a problem. The files could contain an inconsistent set of data with partial transactions. There is also no way to tell when, in Perforce terms, the files were backed up. If subsequent restoration is necessary, the journal file is useless. There would be no way to tell which journaled transactions exist in the backed-up data and where to start replaying the journal file. In this case, it is only possible to restore as of the time of the backup.

This limitation does not exist when using the built-in checkpoint and journal facilities. Using these facilities when recovering from a problem, it is almost always possible to restore the depot to the transaction immediately before the occurrence of the problem. A “journal” counter within the metadata ensures the proper order of file replay.

The versioned file tree

The versioned file tree must be backed up as well, via some external utility. Perforce is resilient when external processes read the versioned file tree. In the rare event that Perforce needs to access a file in the versioned file tree and it is being read by another process, it will retry the operation several times. Always make sure to back up the db.* files with the checkpoint/journal facility before making a backup of the versioned file tree. This ensures that all of the metadata in the backup points to existent revisions in the versioned file tree.

Related commands and utilities

Rsync is an open-source utility that synchronises directory trees from one location to another. It transfers only the pieces of files that change, rather than transferring all parts of all files being copied. This makes it an excellent candidate for use in maintaining a duplicate copy of the Perforce versioned file tree. Information on rsync can be found at this URL: <http://rsync.samba.org/>

Before contemplating backups, it is wise to ensure your db.* files are in a consistent state. The “p4d -xv” command will perform a read-only test of the db.* files. It does a sequential read of the files to ensure each database page is properly connected to the underlying btree structure. It is a good idea to run this occasionally. The command should take a little less time to run than a checkpoint, as it reads all parts of every db.* file that comprises your metadata. It locks users out while it runs. Output of this command looks like this:

Validating db.counters
Validating db.logger
Validating db.user
Validating db.group
Validating db.depot
Validating db.domain
...
Validating db.monitor

Messages about “free list pages which are not marked as free” can be safely ignored. Other warnings in the output indicate serious problems.

Checkpoint/journal file format

In order to better understand how checkpoints and journals work, we should examine the properties of the underlying checkpoint and journal files. This will also help us to understand how to implement more complex backup strategies

The file format and syntax of checkpoint and journal files is identical. The first field is an operation code. The other fields are data fields that tell Perforce what data the operation code is to be applied to. Let’s look at an example:

`@pv@ 1 @db.table@ @data@ 3`

The `@pv@` specifies a “put value” operation. When read, this tells the server to put the values “data” and “3” into the “db.table” database table. The tables are indexed on at least the first data field. The server will take the information provided and correctly identify what to do with the data. It is beyond the purview of this presentation to describe the database schema in more detail.

The available operators are as follows:

<code>@pv@</code>	–Put value	– Inserts data into the database table
<code>@dv@</code>	–Delete value	– Deletes the specified data from the table
<code>@rv@</code>	–Replace value	– Replace existing data with new data
<code>@vv@</code>	–Verify value	– Verify specified counter is set to this value

You will also see these informational records which include the process ID and time:

<code>@ex@</code>	– End transaction
<code>@mx@</code>	– Mid-transaction

These show where the server finished processing one unit of work or where it flushed its cache to disk. There are no “begin transaction” markers.

Ensuring proper replay – File numbering

Checkpoint and journal files are numbered, and the “journal” counter is used to keep track of which numbers are used. It always contains the number of the last checkpoint, which is the same as the number of the next journal. As noted earlier, checkpoints contain a snapshot of the contents of db.* files at a known point in time. Journals contain the transactional changes of the db.* files between checkpoints. When trying to understand the file numbering, remember this equation:

$$\text{Checkpoint.}n + \text{Journal.}n = \text{Checkpoint.}n+1$$

The metadata contained in checkpoint *n* with the application of the transactions contained in journal *n* results in the same metadata as checkpoint *n+1*. To better understand this, look at the following directory listing that has been specially sorted by time and date (but not by filename):

```
08/03/2008 11:23 PM 4,011,575 journal.180.gz
08/03/2008 11:23 PM 175,055,624 checkpoint.181.gz
08/05/2008 06:20 PM 4,775,005 journal.181.gz
08/05/2008 06:20 PM 175,450,056 checkpoint.182.gz
08/07/2008 04:55 PM 12,415,363 journal.182.gz
08/07/2008 04:55 PM 176,253,185 checkpoint.183.gz
08/07/2008 04:55 PM 37 journal
```

In a standard checkpoint operation, journal files are rotated from their unnumbered default of “journal” to numbered files immediately after the checkpoint file is written. From the timestamps in the directory listing above, we can see how the journal files do indeed represent the transactional changes between checkpoints. Now, as an extension of the previous equation, we can also see that this equation holds true as well:

$$\text{Checkpoint.}n + \text{Journal.}n + \text{Journal } n+1 = \text{Checkpoint.}n+2$$

Ensuring proper replay – Internal counters

The “journal” counter is set near the start of a checkpoint. As stated earlier, the data in the checkpoint represents a snapshot of the depot at a point in time, and when replayed it creates a set of db.* files from scratch. This counter setting, then, tells Perforce which journal file comes next. In this snippet from the start of a checkpoint we can see that the next journal is number ten:

```
@pv@ 0 @db.counters@ @change@ 2207
@pv@ 0 @db.counters@ @journal@ 10
@pv@ 0 @db.counters@ @upgrade@ 15
@ex@ 2012 1203543600
...
```

The subsequent journal file looks like this:

```
@vv@ 0 @db.counters@ @journal@ 10
@ex@ 3996 1203947951
@rv@ 3 @db.user@ @Reb@ @Reb@@Reb-ESAU@ @@ 1199913996
    1204644765 @Reb@ @@ 0 @@ 0
@ex@ 3116 1204644765
@rv@ 0 @db.counters@ @journal@ 11
@ex@ 3116 1204644765
```

This journal file resulted from running the following three commands:

```
p4 admin checkpoint
p4 info
p4 admin checkpoint
```

From the end transaction markers, @ex@, we can see that this journal file consists of three transactions. The first transaction consists of the first two lines. It is critical. This set of lines appears at the start of every Perforce journal file. It tells the server reading the file to verify that the value of the “journal” counter is set to 10 before proceeding further. If out-of-order replay is attempted, processing will stop and a message like this will appear:

```
Perforce server warning:
    Journal file 'journal.9' skipped (out of sequence).
```

The second transaction is the replacement of a record within the db.user table. It changes the access time for the “Reb” account. This data is updated only every few minutes, not after every transaction, so it will not always appear. The last transaction indicates the end of the journal file. It uses the @rv@ operator to replace the value of the “journal” counter, indicating that this file was successfully read.

Backup scheduling

There is no one correct schedule for backups. Each installation is different in size, number of users, hardware configuration, and need. Some of the most basic items to consider are the following:

- How much time can I afford to be down?
If a restoration from backup becomes necessary, how long will the overall process take? Too often we see users who concentrate solely on the amount of time a backup process will take, with no attention paid to the overall recovery time. This time includes retrieval of backups from wherever they are stored. If they are not immediately available on the server machine, or if the server machine itself is unavailable, it may take some time to even begin restoration.

An analysis of this point at the start of your planning process will help you to determine what kind of backup solution is best for your site.

- **Checkpoint time**
Checkpoints are not created instantaneously. Creation time depends largely on the amount of data within the db.* files and the speed of your server's I/O subsystem. If compression is used, CPU speed vs. I/O speed will need to be factored into the equation as well.
- **Recovery time**
Checkpoints often contain a large amount of data. While they can be compressed to save space and eliminate I/O during restoration, it is a good idea to consider how long recovery will take.
- **Available storage**
Practical limitations in the amount of storage space available will sometimes determine the amount of checkpoint and journal data to keep backed up. Determine first how big your backed-up dataset is and whether you have allocated enough space.

These items lead to the inevitable questions of how often to back up and how many sets of backups should be kept. Most of our customers back up their installations either every day or every week. Some use backup strategies that allow for recovery on hourly or even more frequent boundaries.

After making sure you have safely kept enough data to recover your depot in the event of a problem, it is okay to delete old backups. The number of backup sets that should be kept can not be quantified, but it can be estimated using this rule:

Keep checkpoint and journal data around long enough that once you realise you have a problem, you can restore from before the time it occurred.

You must be able to recognise that you have no problem with your depot before you delete your older backup data. In most daily checkpoint environments, this means you should keep a minimum of two or three weeks' worth of data. In a weekly checkpoint environment, you should keep a minimum of at least four weeks' worth of data. Generally only the most recent data set or two is needed for restoration. It is always good to be safe, however, and have more. Additional disk space is considerably less expensive than the cost of recovery without proper backups.

It is also a good idea to occasionally check that your backup process is working properly and that the backups are usable. Try replaying them to a test location. This will also give you practice in the event of a real problem. Setting things up and never looking at them again is a recipe for disaster.

What to do when a simple strategy won't work

Now that we have covered the basics of how checkpoints work and how often to take them, we will analyse some more complex strategies. These can be implemented for a variety of reasons, including the following:

- Backup or restore time interferes with business processes:
It takes 3 hours to take a checkpoint and you have a one hour per day maintenance window
- Failover requirement:
You can not afford to have much or any downtime to take a checkpoint
- Other reasons:
Read-only server instance is needed for reporting
Hardware requirements/features

Types of more complex solutions

There are three basic types of more complex solutions to Perforce backup and recovery. Each has its own features and foibles. They are as follows:

- Snapshot checkpoint – Least complex
- Offline checkpoint – More complex
- Near real-time replication – Most complex

Each of these methods works based on the live server's metadata, but each operates on it in very different ways.

Snapshot checkpoint

A snapshot checkpoint is similar to the standard online checkpoint. Instead of locking the live server's db.* files while it runs, however, it works from a copy. This allows for almost no downtime, even when it takes quite a while for the checkpoint to be created. Many operating systems and online disk array subsystems, including those from Network Appliance and EMC, offer a "snapshot" function. It makes a read-only copy of the desired data. Often, such a copy can be made in a fraction of a second even if the dataset is quite large.

In this scenario, the snapshot function of the installed hardware/software is used to create a read-only copy of the server's db.* files at a known point in time. Using some of the commands we have already discussed, we will do the following:

- Tell the server to truncate the current journal
- Lock the Perforce depot and take a snapshot of the db.* files

- Take a checkpoint of the snapshot data

Always truncate the journal so you have a consistent starting point for the next journal. This will allow replay of a checkpoint and multiple subsequent journals if necessary for a restoration. Additionally, since the truncation of the journal and the locking of the depot to take the snapshot occur in two separate commands, something must be done to ensure that no other commands are executed between these two operations. Otherwise, the current journal could contain records from commands that came before the previous checkpoint.

There are a number of ways to do this. The cleanest and easiest way is to shut down the server, rotate the journal and take the snapshot, and restart the server. Starting with the 2007.3 Perforce release, journal rotation on UNIX and UNIX-like systems is instantaneous if journal compression (the “-z” flag) is not used. In this case, Perforce now uses a simple rename instead of a copy. Since snapshot creation is very fast as well, bringing the server down momentarily is the best solution to this problem.

The “journal” counter is incremented when the “p4d -jj” command is executed, and it is important that it not be incremented again during this operation. The journal and the immediately-subsequent checkpoint need to have sequential numbers in order to allow proper playback if they are needed later. So, when checkpointing the snapshot, care must be taken to ensure that the “journal” counter is not changed. The resultant checkpoint must be able to produce a set of db.* files identical to those in the snapshot.

The “p4d -jd *filename*” command takes a checkpoint without affecting the “journal” counter. This command outputs the checkpoint to a specified file. The proper checkpoint number to be used can be found at the end of the newly-rotated journal file or the start of the current journal file. On UNIX/Linux, a small one-line shell script can be used to obtain the appropriate number for use in the filename.

A very basic script that performs these operations might look like this:

```
p4 admin stop # Stop the server
p4d -r /P4ROOT -jj # Rotate the journal
p4d -r /P4ROOT -c "snap create -V perforce P4snap" # Create snapshot
p4d -r /P4ROOT -d # Restart the server
JNL=`head -1 /P4ROOT/journal | \
  grep "@vv@ 0 @db.counters@ @journal@" | \
  sed -e 's/@vv@ 0 @db.counters@ @journal@/' \
  -e 's/@//g'` # Get checkpoint #
p4d -r /P4snap -jd -z /P4offline/checkpoint.$JNL.gz # Checkpoint!
```

In a real-world example, error checking should be performed on each step.

Note that the server root is specified on each server-side command. This ensures the commands are operating on the proper set of db.* files.

Offline checkpoint

An offline checkpoint works from a second set of db.* files and does not rely on external hardware or operating system-specific tools. This procedure rotates and copies the journal file to a second Perforce installation where it is replayed. A checkpoint of this second installation can then be taken, reducing the amount of downtime required on the main server installation. The journal replay and checkpoint operation can be performed on a second server machine which can function as a hot standby server. For this reason, offline checkpointing is quite popular.

To start, a full checkpoint of the main server must be taken. This checkpoint is then copied if necessary (if the offline location is on another machine, for example) and replayed to the offline location. Creating the offline server can be as simple as this:

```
p4d -r /P4ROOT -jc -z          # Create main server checkpoint
p4d -r /P4offline -jr -z filename # Restore to offline location
```

As before, you should always specify the P4ROOT with the “-r” flag to ensure you are reading/writing data to the proper directory.

If the offline server metadata becomes corrupt, it will be necessary to refresh it by taking a fresh checkpoint of the main server. While this may not be a palatable decision to make, it is the best way of ensuring consistency between the two data sets.

Once you have populated your offline server location by replaying the checkpoint, you can begin taking offline checkpoints. This is done by rotating the journal in the main server, replaying the rotated journal file in the offline location, and then checkpointing the offline location. This can be accomplished with the following commands:

```
p4d -r /P4ROOT -jj /P4offline/offline      # Rotate live server's journal
p4d -r /P4offline -jr /P4offline/offline.jnl.33 # Play it into offline data files
p4d -r /P4offline -jd -z checkpoint.34.gz   # Take offline checkpoint
```

As was done in our snapshot checkpoint example, we have used the “p4d -jd” command to take the checkpoint. This leaves the “journal” counter untouched on the offline server and allows subsequent journal files to be read in without generating an “out of sequence” error.

When implementing an offline checkpoint strategy from within a UNIX/Linux shell script, more complex commands than those shown above are needed. You will need to store the journal and checkpoint file name produced by the first commands. These names will be used in the second command. An example of this is as follows:

```

JNL=`p4d -r /P4ROOT -jj /P4offline/offline| \ # Rotate live server's journal
grep Rotating | sed -e 's/Rotating journal to//' -e 's/\.\.\./`
p4d -r /P4offline -jr /$JNL # Play it into offline data files
CKP=`tail -2 /P4offline/$JNL | \ # Get checkpoint number
grep "@rv@ 0 @db.counters@ @journal@" | \
sed -e 's/@rv@ 0 @db.counters@ @journal@//' \
-e 's/@//g`
p4d -r /P4offline -jd -z checkpoint.$CKP.gz # Take offline checkpoint

```

As before, in a real-world example, error checking should be performed on each step.

In a multi-machine offline checkpoint environment, some method of moving the newly-created journal file to the second server machine must be implemented. There are many ways to do this, including using the above-mentioned utility rsync. Details of such operations are beyond the scope of this document.

It is possible to rotate the journal more often than the number of times you take checkpoints. The journal file of the main server can be truncated as often as you wish. Typically this is not done more often than once every hour, though. You could, for example, rotate the journal file with “p4d -jj” once every hour, moving and replaying the journals files each time into the secondary server. Then, once a day, take a checkpoint of the secondary server. Any combination will work as long as the “journal” counter remains consistent between the two servers and each checkpoint operation finishes before the next one starts.

Near real-time replication

Sometimes, it is important to be able to have a near real-time copy of a depot. This could be for a variety of reasons, from disaster recovery to running large read-only operations that would otherwise block users from the main server. While snapshot checkpoints and offline checkpoints will allow you to back up your metadata on a frequent schedule, they do not do so in close to real-time. In these cases, other methods must be used.

Named pipes

On UNIX/Linux systems, it is possible to create a file of a special type called a “named pipe.” This type of file acts as a read/write buffer and shuffles data between processes on a FIFO basis. In this scenario, one server process writes journal entries from a Perforce installation into such a pipe while another reads and replays them into a second installation. To start things off, each must have an identical set of depot files. This necessitates the following steps:

- Shut down the main server.

- Take a checkpoint.
- Replay the checkpoint into the secondary server root.
- Start up both servers, using the named pipe as the journal file/data source.

In practice, there are quite a few limitations that make this method unattractive. If either the reader process or the writer process is not running, the other process will be blocked. Also, the buffer between the writer and reader that is supplied by the operating system is generally very small. If one side writes faster than the other side can read, things will slow down. Finally, since the journal records are written directly into the named pipe, they are not stored in case they are needed later.

While this method certainly can work, there are quite a few pitfalls in its implementation. It is not recommended for use, and it is not officially supported.

p4jrep

A more robust and more supportable solution to the near real-time replication problem (though it, too, is officially unsupported by Perforce) can be found in the Perforce Public Depot. This solution, p4jrep, operates by reading the journal file of the running server and transferring and replaying the journal records to a second server. By relying on the primary server's journal file, the secondary server can be brought up and down as needed without affecting the primary server. Similarly, the primary server can be brought up and down as needed without affecting the secondary server.

As with the other backup/replication methods we have looked at, the second server instance must be populated before starting. In order to set up p4jrep on a server, the following basic steps are necessary:

- Shut down the main server
- Take a checkpoint
- Replay the checkpoint into the secondary server root
- Start up the main server
- Start up the secondary server
- Start up the secondary server p4jrep process

Once the initial set of db.* files is loaded into the secondary server root, starting the replication process is as easy as this command:

```
cd /P4ROOTS/primary
p4jrep p4d -r /P4ROOTS/secondary -f -jr
```

This tells p4jrep to read from the /P4ROOTS/primary journal file and to process those log records into the db.* files located in the /P4ROOTS/secondary directory.

It is important to understand the architecture behind this. The p4d processes for the primary and secondary servers access the db.* files for their respective installations. At the same time, p4jrep reads from the primary server's journal file and calls p4d directly to write data to the secondary server's db.* files. Perforce relies on the underlying operating system for locking, so the p4jrep writes look like just another child process on the secondary server.

As p4jrep plays data records into the secondary server, it records how much of the primary server's journal file has been replicated. This allows either server to be stopped without worry. If the primary server is stopped, records are not replicated to the secondary server until the primary server is restarted and additional records are added to its journal. If the secondary server is checkpointed, p4jrep will be forced to wait to update that installation's db.* files, just like any other process attempting to access them. As soon as the checkpoint operation finishes, the necessary journal records will be replayed.

Because p4jrep operates as a separate process, if the secondary server is stopped, p4jrep will still be able to access the db.* files in that installation. If you need to completely shut down the secondary installation, you must bring down both the secondary p4d server and the p4jrep process.

This replication method requires only one special tool, p4jrep itself. The journal file is written to by one server and read from by another, and standard files are used. Either server can be checkpointed using the tools built into Perforce. When the operation is finished, p4jrep will continue replicating data into the secondary server from where it left off. One wrinkle to remember is that p4jrep must be running whenever you rotate the journal file on the main server.

If you use p4jrep as a method of offline checkpointing, you must be careful when checkpointing the main server. When checkpointing the secondary server, its "journal" counter will be incremented. If the primary server's "journal" counter is subsequently incremented because of a checkpoint or journal operation, it will overwrite the updated value in the secondary server. A subsequent checkpoint in that server would have the same checkpoint number and could cause confusion or even overwrite an existing checkpoint file.

There are several ways around this problem. The simplest is to wait until the end of the checkpoint, and the journal rotation of the secondary server. Then, run the "p4 counters" command to obtain the value of the "journal" counter in the secondary server. Finally, use that value to update the same counter in the primary server. If the secondary server is running on port 1667 and the primary is on port 1666, you could automate such an update like this:

```
JNL=`p4 -p 1667 counters | grep journal | \      # Get counter value from
      sed -e 's/journal = //'             # secondary server
p4 -p 1666 counters journal $JNL        # Set counter in primary
```

As with our other examples, in a real-world scenario, error checking is a must. Be extremely careful when setting Perforce's internal counters like this. Improper settings can have unpleasant consequences. To ensure you are careful, the above example requires the addition of a "-f" flag in the command that actually sets the counter.

While this method is listed above as "most complex," it really does not have to be. P4jrep in its basic form is quite easy. More complex implementations are certainly possible. The above assumes that you would like the replicated, secondary installation to remain identical to the primary one. By default, p4jrep replicates everything. This is not always desired. For this reason, p4jrep supports filters. Using a simple egrep command, it is possible to filter out any kind of journal record that is not desired. For example, to filter out db.have records, create a shell script, egrepfilter.sh, like this:

```
#!/bin/sh
# Filter out named tables by excluding the journal entries.
egrep -v '@db\.have@ ' $1 | p4d -r /P4ROOTS/secondary -f -jr -
# END
```

Then, start p4jrep using the filter, like this:

```
p4jrep egrepfilter.sh
```

All db.have entries will then be stripped from the data feed before they are replicated into the secondary server.

Some users of p4jrep modify the replicated data in other ways, deleting client specifications, performing updates, etc. In these situations, the primary and secondary data sets diverge. Backup strategies for each of these data sets are likely to be different. In this case, offline or snapshot checkpoints of the primary and secondary data might be in order.

Summary

Perforce offers a variety of tools for use in backing up and replicating your data. These can be combined with operating system features and other tools to suit a wide variety of purposes. An understanding of how these pieces work, from the most basic levels, should allow you to combine them to meet your business needs. Mix and match them as you see fit, but please use them! If you have any questions, call Perforce Technical Support. We are always happy to assist.