Base Picking

Sam Stafford

Perforce Software, Inc

A deep dive into the integration engine

Table of Contents

The function of the base in a three-way merge

A three-way merge operation combines the changes from two files (the "legs") relative to a common origin (the "base").

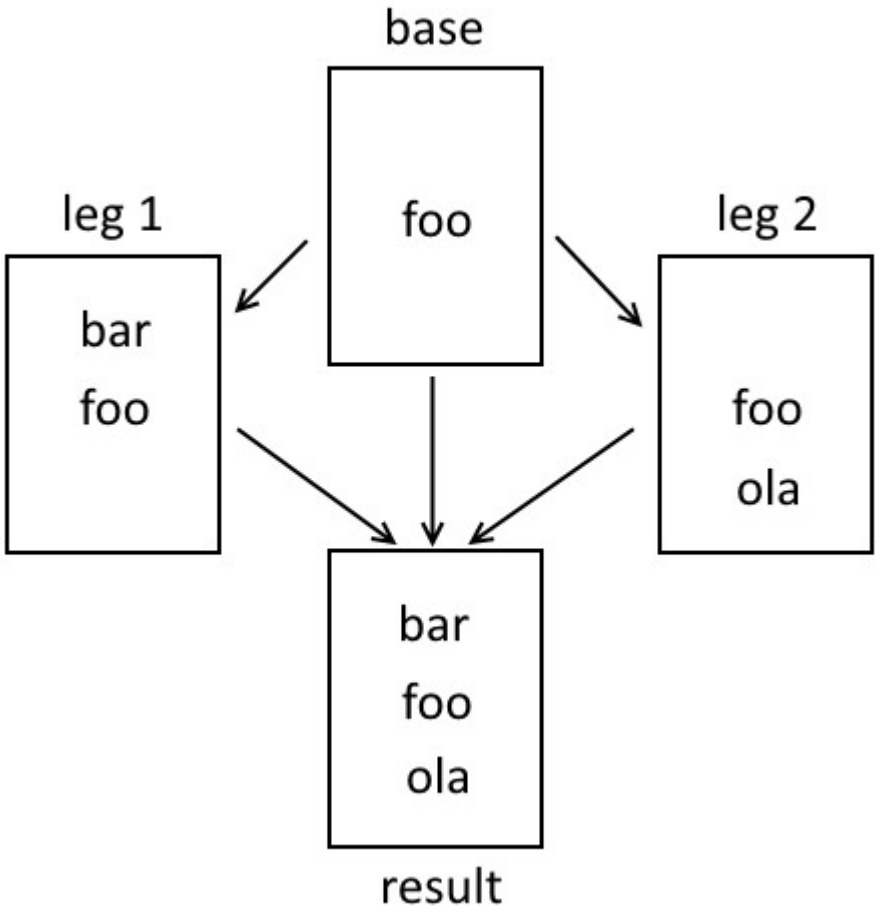

Figure 1: a three way merge where both legs add content

Figure 1 illustrates a simple example of a textual three way merge. The delta (difference) between the first leg and the base is the text "bar", inserted at the beginning of the file, and the delta between the second leg and the base is the text "ola", inserted at the end of the file. The merged result contains the union of both deltas.

Although in figure 1 the merged result also contains the union of the *content* from both legs, this is not necessarily always the case; the role of the base in a three-way merge is to determine the delta, which in turn determines the result. An example of a "negative" delta (removing content, rather than adding it, relative to the base) is given in figure 2.

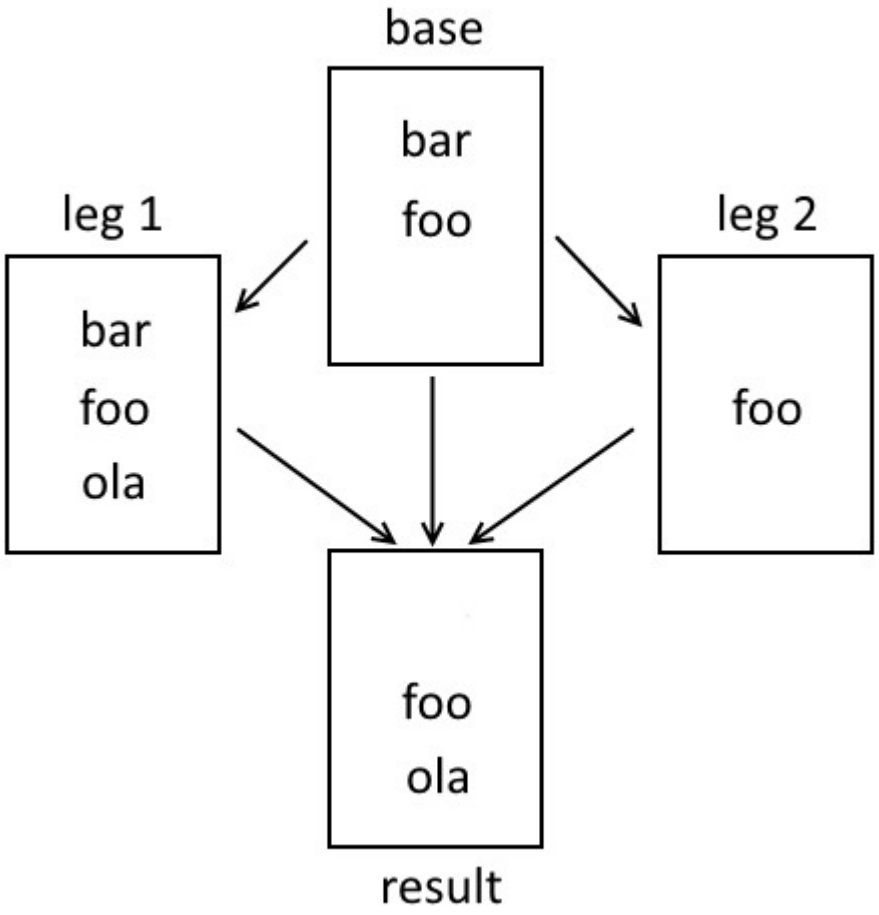

Figure 2: a three way merge where one leg removes content

In a textual merge, it is possible for the deltas to overlap with one another; when this happens, it may not be possible for a fully automated merge process to determine the correct result, and a "conflict" is generated requiring human intervention. To the extent it is possible to do so while still producing the desired result, it is desirable to perform a merge using a base that minimizes the deltas, since minimizing the deltas minimizes the probability of their overlapping, and therefore the probability of a conflict.

Although in practice the subject of a merge operation is typically a set of text files, the concept of combining differences can be extended to other types of data, such as binary sets, which make it possible to model merge interactions as a truth table. Note that in each column, the difference(s) between the base and the two legs determine the final result.

Base	0	0	0	0	1	1	1	1
Leg 1	0	0	1	1	0	0	1	1
Leg 2	0	1	0	1	0	1	0	1
Result	**0**	**1**	**1**	**1**	**0**	**0**	**0**	**1**

By imagining that each column of this table corresponds to a particular section of a text file, with "1" corresponding to a particular piece of text being present and "0" corresponding to its absence, we could map this model back to a non-conflicting textual merge. Modeling a merge as a more abstract set of elements rather than as actual text files provides a number of advantages when dealing with large numbers of changes, though, so this paper will predominantly discuss merges in this more abstract form rather than providing textual examples.

Modeling a file's revision graph as a growing set of elements

Each revision of a file represents some change to that file's content. By representing each change as a new element being added to a set that represents a file's contents, commonalities between revisions can be modeled independently of the actual content.

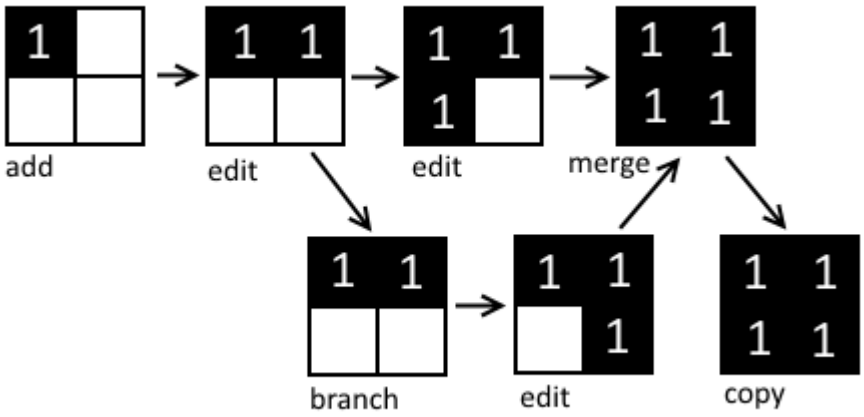

Figure 3: Branched file history as a set of elements

Figure 3 represents different revisions of a file as different sets of up to 4 unique elements, with each element corresponding to one of four positions in a grid; each element is either absent or present within a given revision. The initial "add" revision contains only the first element, and subsequent "edit" revisions each add a new element, while also including any elements present in the previous revision. When a "branch" is created, it is identical to the revision that it is branched from, and when the two branched variants are "merged", the resulting revision contains all the elements of both.

Note that the base of this merge would be the revision that was initially branched; hence the edit in each of the legs is a delta that becomes part of the result. It is intuitive that in this simple case the branch point is the "best" possible base, but to handle more complex cases, we require a rigorous definition of the best base that can be found in any given situation.

Finding the best merge base within a revision graph

The purpose of a merge operation is to propagate changes from one file to another. One leg of the merge is the target file, and the other is the source file, which contains the set of changes to be propagated. The desired result of the merge is the target file with the source changes applied to it.

For the merge base to produce the desired result, each element that is different between the two legs must be absent from the base if it is to be present in the result, and present in the base if it is to be absent from the result.

For the merge base to minimize the chances of conflict during a real-world merge, each element that is identical between the two legs should also be identical with the base.

Given a graph of each existing revision of a file, and a known result that we are trying to produce by merging two of those revisions, the set of acceptable bases is defined as those that will in theory produce the desired result, and the best possible base is the base that within the set of acceptable bases will minimize the odds of a merge conflict.

Modeling the effect of different resolve and submit actions

Creating an abstract model of the contents of each revision according to the revision and integration metadata allows us to simulate past merges and determine the optimal base for a pending merge without needing to access the actual content -- given that a merge operation may easily include millions of files, this scalability is vital. For debugging purposes a textual representation of this model is included in the server log when the **dmc=5** trace flag is enabled. Revision and integration metadata for individual files can be accessed via the client **p4 filelog** command, or viewed visually via the Revision Graph graphical tool.

Each "edited" revision adds one new element of content (hereafter an "edit") to the set that represents the file's history. A file that is simply added and edited, with no branching or merging, will simply accrue one edit per revision. A revision created as a result of an integration operation is considered "edited" if the operation is one that permits edits to be made, such as a "move" command or an "edit from" resolve action or a "branch" action that is converted into an "add" action. An integration operation in which the target may contain no changes other than those propagated from the source is considered a "pure" integration and does not add a new edit.

If an integration record indicates that the source and target revisions are identical (a "branch" or "copy"), then the abstract model reflects this and sets the target revision to be identical to the source. Although in most cases a given revision will include all of the edits of the previous revision, a "copy" operation can cause edits to be effectively removed from a file.

If an integration record indicates a merge (either a pure "merge into" or an edited "edit into"), then the range of edits corresponding to the source revision range of the integration record is combined with the previous revision of the target to produce a simulated merge result, which is then taken to be the contents of that revision. If multiple merges were performed into a single revision, they are added together, and if an "edit into" is involved, one additional unique edit is introduced as well.

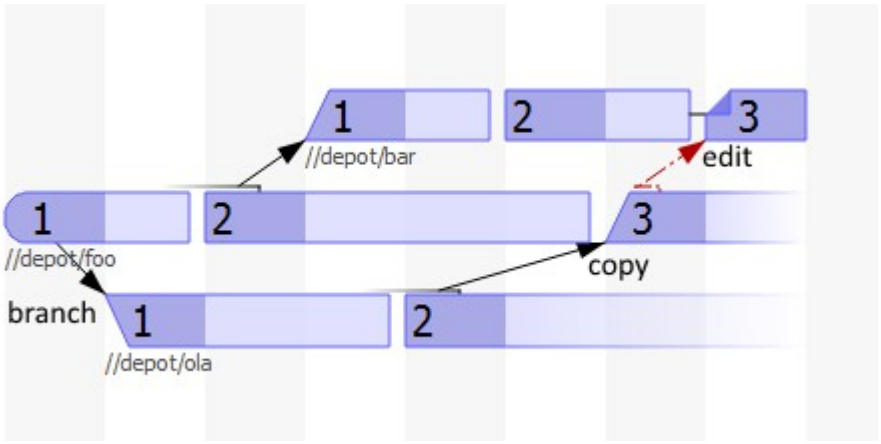

Figure 4: copy and merge actions

The scenario illustrated in figure 4 is represented in the content model as follows:

```
1.... //depot/foo#1
1.... //depot/ola#1
11... //depot/foo#2
11... //depot/bar#1
1.1.. //depot/ola#2
11.1. //depot/bar#2
1.1.. //depot/foo#3
1.111 //depot/bar#3
```

Each column represents the presence or absence of a particular edit, with "1" indicating presence and the placeholder "." indicating absence. The two revisions that are particularly of note in this graph are foo#3 and bar#3.

The copy from ola#2 to foo#3 results in two differences between foo#2 and foo#3. One edit is removed (the edit originating in foo#2) and one edit is added (the edit originating in ola#2):

```
11... //depot/foo#2
1.1.. //depot/foo#3
```

The edited (or "dirty") merge from foo#3 to bar#3 propagates these differences while also adding a new edit:

```
11.1. //depot/bar#2
1.111 //depot/bar#3
```

Note to the reader: if this section doesn't make sense yet, it's just going to get more confusing from here.

Tracking ignored edits

An "ignored" integration record creates persistent divergence between two files -- in other words, it specifies that there are some changes that belong in one branch of a file but not another, and it requests that future merge operations preserve that relationship for those changes while continuing to propagate newer changes.

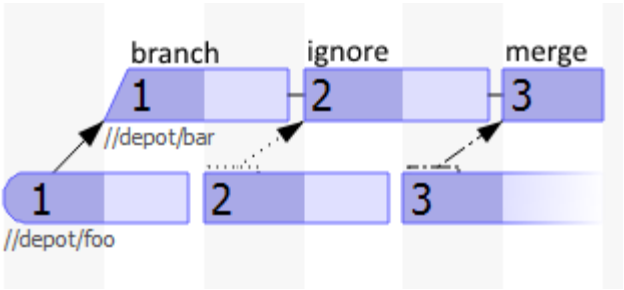

Figure 5: an ignore followed by a merge

```
1.. //depot/foo#1
1.. //depot/bar#1
11. //depot/foo#2
1i. //depot/bar#2
111 //depot/foo#3
1i1 //depot/bar#3
```

The "i" in the content model represents an ignored edit. This edit is not actually present in the file, but the "i" acts as a placeholder that prevents future merge operations from attempting to add this particular edit to this particular file.

Since merges can propagate the removal of an edit just as easily as the addition of an edit, the model includes for the possibility of a removal being ignored.

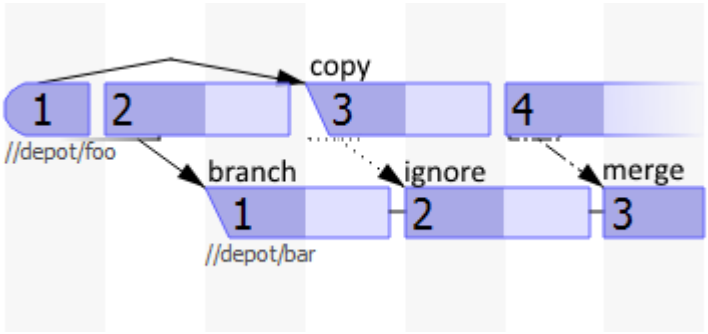

Figure 6: an ignored removal

```
1.. //depot/foo#1
11. //depot/foo#2
11. //depot/bar#1
1.. //depot/foo#3
1I. //depot/bar#2
1.1 //depot/foo#4
1I1 //depot/bar#3
```

The "I" represents an edit that is present but with an ignored removal -- that is, future merge operations should not attempt to remove this edit.

Determining the result of an integration

The integration engine receives as input a range of revisions for a particular depot file (the source) and the path of a file that can be opened on the client (the target), and returns as its output a scheduled resolve that will propagate the changes from the source revisions into the open target file (or no resolve, in which case the file is not opened at all). The source revision(s) and target path for each input file are determined by the branch view and revision range supplied to the **p4 integrate** (or **p4 merge**) command.

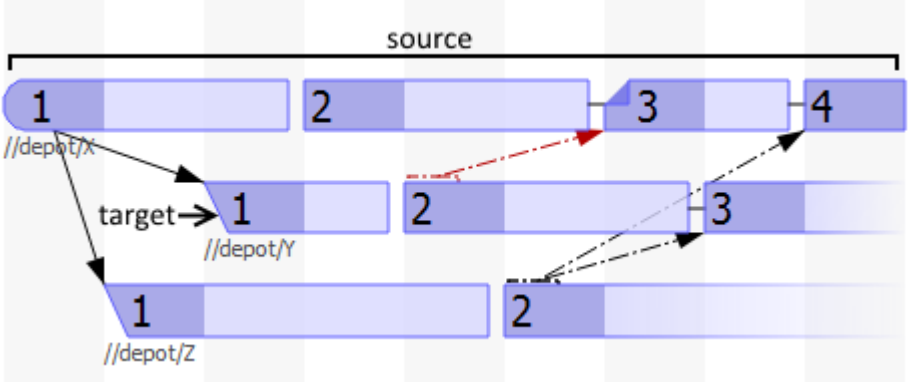

Figure 7: setting up an integration from X to Y

The **dmc=5** server configuration setting produces detailed trace logging when we run an integration from X to Y as illustrated in figure 7. The integration operation starts with the source revision range and the target:

```
==integ //depot/X#1,#4 -> //depot/Y==
```

Integration credits are loaded in order to build the graph:

```
credit //depot/X#1,1 -> //depot/Y#1,1 (branch from)
credit //depot/X#3,3 <- //depot/Y#2,2 (edit into)
credit //depot/Z#2,2 -> //depot/Y#3,3 (merge from)
credit //depot/Y#1,1 <- //depot/X#1,1 (branch into)
credit //depot/Y#2,2 -> //depot/X#3,3 (edit from)
credit //depot/Z#1,1 <- //depot/X#1,1 (branch into)
credit //depot/Z#2,2 -> //depot/X#4,4 (merge from)
credit //depot/X#1,1 -> //depot/Z#1,1 (branch from)
credit //depot/X#4,4 <- //depot/Z#2,2 (merge into)
credit //depot/Y#3,3 <- //depot/Z#2,2 (merge into)
```

The content of all the files in the graph is modeled as follows, with "pure" (unedited) revisions marked with a "p" and revisions that were created by "copying" another revision marked with a "c":

```
1.............................. ----- //depot/X#1
1.............................. pc--- //depot/Y#1
11............................. ----- //depot/Y#2
1.............................. pc--- //depot/Z#1
1.1............................ ----- //depot/Z#2
111............................ p---- //depot/Y#3
1..1........................... ----- //depot/X#2
11.11.......................... ----- //depot/X#3
11111.......................... p---- //depot/X#4
```

The result of the X->Y integration is determined by walking through the history of X and Y and accumulating a list of edits from X that may need to be applied to Y (the "debit"):

```
===debit calculation for //depot/X#1,#4===
===trimming debit to //depot/X#2,#4 via copy===
//depot/Y#1 copied from //depot/X#1
1.............................. ----- //depot/X#1
1.............................. pc--- //depot/Y#1
11............................. ----- //depot/Y#2
111............................ p---- //depot/Y#3
1..1........................... ----- //depot/X#2
   +                           ----- accumulated debits
11.11.......................... ----- //depot/X#3
 + ++                          ----- accumulated debits
11111.......................... p---- //depot/X#4
 ++++                          ----- accumulated debits
11111.......................... ----- source
 ++++                          ----- debit
                               ----- ignored (source+target)
111............................ ----- target
11111.......................... ----- result
```

Certain heuristics may be used to trim the debit prior to this calculation, and this will be noted in the trace log; in this case the fact that X#1 was copied (branched) into Y#1 means that X#1 does not need to be included in the debit.

The debits are accumulated by comparing each revision in the source range to the revision before it and tracking the set of edits that were added or removed. This set is then applied to the target to produce the result -- that is, the content of the revision that would be created in an ideal pure merge from the source to the target. If at this point the result already looks like the head revision of the target file, no merge needs to happen at all.

Picking the base

Continuing with the same example, the integration engine now uses the merge result (calculated in the previous step) and the two legs of the merge (the source and target revisions, which are also known at this point) to determine what contents the base should have.

```
===begin base search===
11111........................... ----- theirs
111............................. ----- yours
11111........................... ----- result
   ++                           ----- to insert/remove
111............................. ----- ideal base
1............................... ----B //depot/X#1
1............................... pc--b //depot/Y#1
11.............................. ----B //depot/Y#2
1............................... pc--b //depot/Z#1
1.1............................. ----b //depot/Z#2
111............................. p---B //depot/Y#3
1..1............................ ----- //depot/X#2
11.11........................... ----- //depot/X#3
11111........................... p---- //depot/X#4
best base: //depot/Y#3
```

The two legs of the merge are "theirs" and "yours" (corresponding to the **p4 resolve** command that the integration engine is setting up). For each edit that differs between "theirs" and "yours", we need the base to also differ from the desired result, since a merge always applies the difference relative to the base. For each edit that is identical between "theirs" and "yours" (and which should also be identical with the result), an ideal merge will produce the correct result regardless of the state of the base, but having that edit be identical in the base as well will reduce the chance of a conflict, so this is preferred if possible.

Using these rules it is easy to determine what the ideal base should look like, and also to establish a function for what an "acceptable" base must contain (or exclude) and how much a given acceptable base differs from the ideal. The graph is traversed and each revision is scored according to this function. A "B" in this section indicates that the revision is the best base seen so far; a "b" indicates that the revision is an acceptable base but is not the best so far.

In this example, it happens that there is an actual revision matching the ideal base (Y#3) and so this is the chosen base for the merge.

Picking a non-ideal base

In the previous example, a revision was found that exactly matched the calculated ideal base. In this example, the ideal base does not exist, so it is necessary to find the best acceptable base from among the existing revisions.

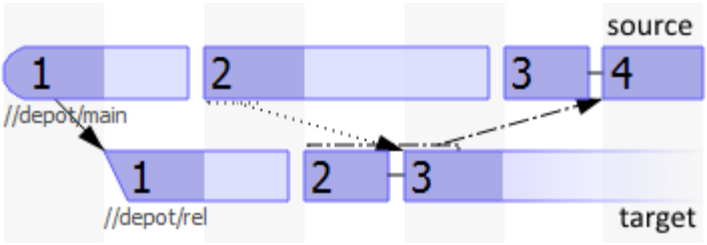

Figure 8: integrating around an ignored edit

```
===debit calculation for //depot/main#1,#4===
===trimming debit to //depot/main#2,#4 via copy===
...applied reverse credit to //depot/main#4
...applied forward credit to //depot/main#2
//depot/rel#1 copied from //depot/main#1
===trimming debit to //depot/main#3,#4 via direct credit===
1.............................. pc--- //depot/rel#1
11............................. ----- //depot/rel#2
1.1............................ ----- //depot/main#2
11i............................ p---- //depot/rel#3
  i                            ----- accumulated ignores
1.11........................... ----- //depot/main#3
   +                           ----- accumulated debits
1111........................... p---- //depot/main#4
 + +                           ----- accumulated debits
1111........................... ----- source
 + +                           ----- debit
  i                            ----- ignored (source+target)
11i............................ ----- target
11.1........................... ----- result
```

The edits that are ignored in the target are tracked along with the debits that have been added to the source, and factor into the final result. In this example main#2 is automatically trimmed out of the debit due to having been directly integrated into the target, but if it had not, the ignore would be used to "mask out" that edit and the calculated result would be the same.

```
===begin base search===
1111.............................. ----- theirs
11i............................... ----- yours
11.1.............................. ----- result
   +                              ----- to insert/remove
111............................... ----- ideal base
1................................. ----B //depot/main#1
1................................. pc--b //depot/rel#1
11................................ ----B //depot/rel#2
1.1............................... ----B //depot/main#2
11i............................... p---b //depot/rel#3
1.11.............................. ----- //depot/main#3
1111.............................. p---- //depot/main#4
best base: //depot/main#2
```

The ideal base in this case is a revision that does not exist. In scoring the potential bases, main#2 is favored as the best available option because it contains the edit that was ignored -- although the base selection process attempts to include as many of the ideal base's edits as possible, we prioritize the ignored edits, because including these will actually change the merge result even in an ideal situation, whereas including a common edit may increase the chances of a conflict but will not change the outcome of an ideal merge.

When resolving with this base, the user will be re-resolving the edit from rel#3 (which was previously merged into main#4), but will not be bringing the previously ignored main#2 edit into rel.

Propagating a negative debit

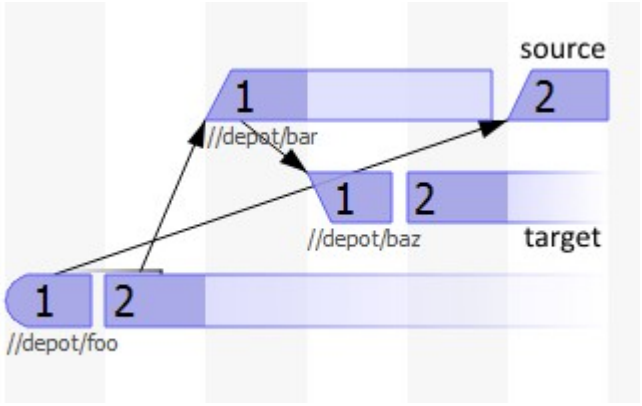

Figure 9: a source with removed edits

In figure 9 we have a source file that has had edits removed via a copy from an older revision. The trace logging is hopefully mostly self-explanatory by this point:

```
===debit calculation for //depot/bar#1,#2===
===trimming debit to //depot/bar#2 via copy===
//depot/baz#1 copied from //depot/bar#1
11.............................. pc--- //depot/bar#1
11.............................. pc--- //depot/baz#1
1............................... pc--- //depot/bar#2
 -                                ----- accumulated debits
1............................... ----- source
 -                                ----- debit
11.............................. ----- target
1............................... ----- result
===begin base search===
1............................... ----- theirs
11.............................. ----- yours
1............................... ----- result
 -                                ----- to insert/remove
11.............................. ----- ideal base
1............................... ----- //depot/foo#1
11.............................. ----B //depot/foo#2
11.............................. pc--B //depot/bar#1
11.............................. pc--b //depot/baz#1
1............................... pc--- //depot/bar#2
best base: //depot/bar#1
```

The only new thing in this example is the negative debit, which is generated by virtue of bar#2 having one fewer edit than bar#1. The negative debit is applied to the target to generate the result, and from that point the base calculation happens exactly as with the previous examples.

In a more complex example we could easily have a mix of positive and negative debits and ignores, since each edit is tracked individually across the entire graph.

Remapping renamed files

The examples so far have been cases where the source and target file both exist and are both related. If one or other of the files have been renamed, the target file generated by mapping the source through the branch view will probably not be the file that actually has common ancestry with the source.

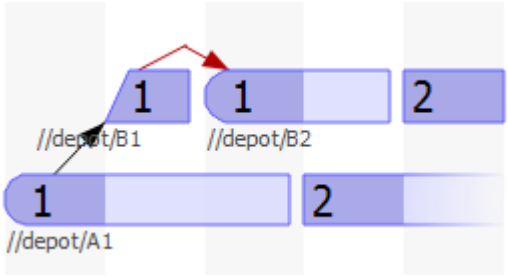

Figure 10: integrating a renamed file

In figure 10, **//depot/B*** is being integrated to **//depot/A***. The desired outcome in this situation is for resolve to be able to merge the edits from B2 and A1, and to move A1 (including the merge result) to A2. The integration engine skips B1 because its head revision is a "move/delete", depending on the fact that every "move/delete" will have a matching "move/add", which in this case is B2.

The initially mapped target when integrating B2 is A2, which does not exist. If the target file does not exist, or if the base search process determines that the existing target file has nothing in common with the source file, the graph is searched for a better match.

```
==integ //depot/B2#1,#2 -> //depot/A2==
===searching for credit into //depot/A2===
Checking for match between //depot/B2#2 and //depot/A2#none:
    checking //depot/B2#2 for move...
    ... moved //depot/B[1->2]
    testing branch ancestry of //depot/B1#1 and //depot/A1#1
      branch history B1#1 <- B1#1 <- A1#1
      branch history A1#1 <- A1#1
    common ancestor: //depot/A1#1 (move diff 0)
```

The move from //depot/B1 to //depot/B2 is used to translate //depot/A2 backward to the path //depot/A1. The histories of B1 and A1 are then both explored backwards to see if they converge at a common ancestor -- since we are only trying to determine at this point if they are the "same" file rather than to identify precise points of commonality and divergence, this is a much simpler process than the base calculation.

Since the two do converge, the base selection process is restarted with the new target path:

```
===searching for credit into //depot/A1===
===begin base search===
111............................. ----- theirs
1............................... ----- yours
111............................. ----- result
 ++                             ----- to insert/remove
1............................... ----- ideal base
1............................... ----B //depot/A1#1
1............................... pc--b //depot/B1#1
11.............................. -c-m- //depot/B2#1
111............................. ----- //depot/B2#2
best base: //depot/A1#1
move base is //depot/A1
```

The "move base" is determined based on the common ancestor found during the matching process and is used for the filename resolve, which will determine how the open target file (A1) is to be moved as part of the resolve operation. This is very similar to a content merge, except that the filenames themselves are merged to produce the result. The "theirs" filename is the originally mapped path, and "yours" is (as always) the open file. Hence:

Base: **//depot/A1**
Theirs: **//depot/A2**
Yours: **//depot/A1**
Result: **//depot/A2**

Incrementing the move base

In cases involving more than one rename, there are multiple potential move bases, just as there are multiple revisions that might serve as a potential base when merging the content. In the following example, the move base is incremented in order to simultaneously ignore one move action while propagating another.

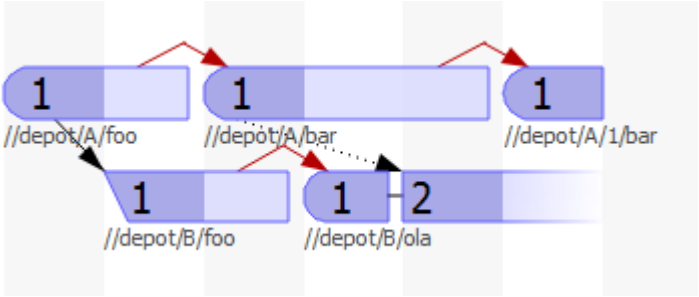

Figure 11: an ignored move

The file in A has been moved twice, once from "foo" to "bar" and once from "bar" to "1/bar". The file in B has been moved from "foo" to "ola". When we integrate from A to B, the source file is **A/1/bar** and the initial target is **B/1/bar**. Since there is no target file, we look for a better match and discover **B/ola**:

```
==integ //depot/A/1/bar#1 -> //depot/B/1/bar==
===searching for credit into //depot/B/1/bar===
Checking for match between //depot/A/1/bar#1 and
//depot/B/1/bar#none:
    checking //depot/A/1/bar#1 for move...
    ... moved //depot/A/[bar->1/bar]
    checking //depot/A/bar#1 for move...
    ... moved //depot/A/[foo->bar]
    testing branch ancestry of //depot/A/foo#1 and //depot/B/foo#2
      branch history A/foo#1 <- A/foo#1
      branch history B/foo#2 <- B/foo#1 <- A/foo#1
    common ancestor: //depot/A/foo#1 (move diff 0)
CCA: //depot/A/foo#1
    checking //depot/B/foo#2 for move...
    ... moved //depot/B/[ola->foo]
    checking //depot/B/ola#1 for move...
===searching for credit into //depot/B/ola===
```

The **dmc=5** trace log shows the path of the search from the initial source file (**A/1/bar**) backward to the point where the source file has a corresponding target file (**foo**) and then forward to the current version of the target file (**B/ola**).

Once the new target file has been found, the debit and base calculation can happen. The results of the base calculation for the content are used to improve the move base:

```
===begin base search===
1.11........................... ----- theirs
11i............................ ----- yours
11.1........................... ----- result
   +                           ----- to insert/remove
1.1............................ ----- ideal base
1.............................. ----B //depot/A/foo#1
1.............................. pc--b //depot/B/foo#1
11............................. -c-m- //depot/B/ola#1
1.1............................ -c-mB //depot/A/bar#1
11i............................ p---- //depot/B/ola#2
1.11........................... -c-m- //depot/A/1/bar#1
best base: //depot/A/bar#1
move base is //depot/B/foo
looking for latest credited move/add...
move/add //depot/A/bar#1 already credited
incrementing base to //depot/B/bar
```

The move base starts at the common filename **foo**, and additional move bases are examined by following the move actions that have happened in the source (searching forward from the common point) and mapping them to the target.

The first source move was from **foo** to **bar**, so we check to see if the result of that move (//**depot/A/bar#1**) has "credit" into the target; in this context that is determined by whether it was flagged as "best base so far" during the base calculation. It was (indicated by the capital B in the base search log), so the move base is incremented from **foo** to **bar**, making the base for the filename merge //**depot/B/bar**. The resulting filename merge produces a new path that merges **[bar->1/bar]** into the existing **ola** path.

Base: //**depot/B/bar**
Theirs: //**depot/B/1/bar**
Yours: //**depot/B/ola**
Result: //**depot/B/1/ola**

Appendix: Esoteric configurables

The following configurable settings are undocumented and are described in this paper for educational purposes. They are not recommended for use on production servers, but may be useful when experimenting with different integration scenarios in a test environment. Configurables may be enabled via the "p4 configure" command, e.g.:

```
p4 configure set dmc=5
p4 configure unset dmc
```

The **dmc** configurable corresponds to passing the flag **-vdmc=N** on the **p4d** command line, or setting **P4DEBUG=dmc=N**. Setting **dmc** to higher thresholds enables trace logging for different server commands; at a level of 5 or higher, the log will include the integration engine logging seen in this paper. Due to the high volume of data written to the log, this is not recommended for use on production servers.

The **dm.integ.tweaks** configurable may be set to enable specific modications to the integration engine behavior. Different tweaks can be combined by adding them. The following tweaks are available as of the 2016.1 release (this text is taken from **p4 help undoc**):

```
dm.integ.tweaks  0 Modify integrate behavior (engine=3 only):
                 1: Treat all 'copy' records as 'merge'
                 2: Treat all 'ignore' records as 'edit'
                 4: Retain credit for copied-over edits
                 8: Force convergent merge of all edits
                16: Legacy (pre-2011.1) resolve behavior
```

The **1** and **2** tweaks both enable simple transformations of integration records; **tweaks=1** causes all "copy" records to be read as if they were "merge", and **tweaks=2** causes all "ignore" records to be read as if they were "edit". These very simple changes have wide-ranging effects. For example, if there are no copy records, edits will never appear to have been rolled back, and if there are no ignore records, it will not be possible to preserve divergence when merging back and forth between branches.

The **tweaks=4** setting modifies the way that the content model is built, by causing "copy" records to always be considered additive. This can produce behavior similar to **tweaks=1** except by a different mechanism; it is also more similar to previous versions of the integration engine in the sense that once a revision has "credit" it retains it in spite of subsequent operations that roll back that revision's content.

The **tweaks=8** setting modifies the debit calculation, simply setting the result to be the union of the source and target, rather building the result through accumulation of positive and negative debits and masking ignored edits. This setting is automatically disabled when setting a source revision range (cherry-picking).

The **tweaks=16** setting disables any type of resolve that did not exist prior to 2011, including (but not limited to) filetype resolves, filename resolves, and true baseless merges.