The Rsync Algorithm

This paper describes the rsync algorithm, which provides a nice way to remotely update files over a high latency, low bandwidth link. The paper itself concentrates on the core algorithm, giving the basic mathematical justifications and characterising the problem. In my presentation at OLS I will concentrate on some of the practical applications of the rsync algorithm, such as the rsync tool and recent work on getting the rsync algorithm into a variety of everyday protocols, such as CVS and HTTP.

1. Notes

1.1. Original presentation

The original presentation of this talk occurred in room C of the Ottawa Linux Symposium, Ottawa Congress Centre, Ottawa, Ontario, Canada on the 21st of July, 2000 at 15:15 local time. This presentation was given by Dr. Andrew Tridgell.

1.2. Presenter bio

Prior to joining Linuxcare, Andrew held a full-time position as a researcher and lecturer in the Department of Computer Science at Australian National University in Canberra. His research interests include parallel computing, network protocols, automatic speech recognition, and operating systems. Andrew spends his spare time working on various bits of free software. He is well known as the original author and current team leader for the Samba software package.

In addition to Samba, Andrew is recognized for his work on rsync, a fast file transfer program; Jitterbug, a Web-based bug tracking system; Nightcap, a learning chess program; and AP/Linux, a Linux port to the AP1000+ multicomputer. Andrew is also one of the privileged few to have witnessed the famous incident where Linus Torvalds was bitten by a penguin.

1.3. Presentation recording details

This transcript was created using the OLS-supplied recording of the original live presentation. This recording is available from ftp://ftp.linuxsymposium.org/ols2000/2000-07-21_15-02-49_C_64.mp3

The recording has a 64 kb/s bitrate, 32KHz sample rate, mono audio (due to the style of single microphone recording used) and has a file size of 44925120 bytes. The MD5 sum of this file is: b790b88d23f2815c823ba958840b72a4

1.4. Creation of this transcript

1.4.1. Request for corrections

This transcript was not created by a professional transcriptionist; it was created by someone with technical skills and an interest in the presented content. There may be errors found within this transcript; we ask that you report them to using the bug tracking interface described at http://olstrans.sourceforge.net/bugs.php3

1.4.2. Tools used in transcript creation

This transcription was made from the MP3 recording of the original presentation, using XMMS for playback and lyx (with docbook template) for the transcription.

1.4.3. Format of transcript files

The transcribed data should be available in a number of formats so as to provide more ready access to this data to a larger audience. The transcripts will be available in at least HTML, SGML and plain ASCII text formats; other formats may be provided.

1.4.4. Names of people involved with this transcription

This transcript was created by Jacob Moorman of the Marble Horse Free Software Group (whose pages live at http://www.marblehorse.org). He may be reached at roguemtl@marblehorse.org

The primary quality assurance for this document was performed by Stephanie Donovan. She may be reached at sdonovan@achilles.net

1.4.5. Notes related to the use of this document

This document is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. While quality assurance checks on this transcript were performed, it was not created nor checked by a professional transcriptionist; the technical accuracy of this transcript is neither guaranteed nor confirmed. Please refer to the original audio recording of this talk in the event confirmation of the speaker's actual statements are needed.

1.4.6. Ownership of the content within this transcript

These transcripts likely contain content owned, under copyright, by the original presentation speaker; please contact them for licensing requests, but do so in a polite manner, please. It may also be useful to contact the coordinator for the Ottawa Linux Symposium, the original venue for this presentation. All trademarks are property of their respective owners.

1.5. Markup used in this transcript

1.5.1. Time markers

At the end of each paragraph within the body of this transcript, a time offset is listed, corresponding to that point in the MP3 recording of the presentation. This time marker is emphasized (in document formats in which emphasis is supported) and is placed within brackets at the very end of each paragraph. For example, *[05m, 30s]* states that this paragraph ends at the five-minute, thirty-second mark in the MP3 recording.

1.5.2. Questions and comments from the audience

These recordings were created using a bud microphone attached to the speaker during their presentation. Due to the inherent range limitations of this type of microphone, some of the comments and questions from the audience are unintelligible. In cases where the speaker repeats the audience question, the question shall be omitted and a marker will be left in its place. Events which happen in the audience shall be bracketed, such as: [The audience applauds.]

Further, in cases where the audience comments or questions are not repeated by the speaker, they shall be included within this transcript and shall be enclosed within double quotes to delineate that the statements come from the audience, not from the speaker.

1.5.3. Editorial notes

The editor of this transcript, the transcriptionist (if you will), and the quality assurance resource who have examined this transcript may each include editorial notes within this transcript. These shall be placed within brackets and shall begin with 'ED:'. For example: [ED: The author is referring to sliced cheese, not grated cheese.]

1.5.4. Paragraph breaks

The paragraph breaks within this transcript are very much arbitrary; in many cases they represent pauses or breaks in the speech of the speaker. In other cases, they have been inserted to allow for enhanced clarity in the reading of this transcript.

1.5.5. Speech corrections by the speaker

During the course of the talk, the speaker may correct himself or herself. In these cases, the corrected speech will be placed in parenthesis. The reader of this transcript may usually ignore the parenthised sections as they represent corrected speech. For example: My aunt once had (a dog named Spot, sorry) a cat named Cleopatra.

1.5.6. Unintelligible speech

In sections where the speech of the author or audience has been deemed useful, but unintelligible by the transcriptionist or by the quality assurance resource, a marker will be inserted in their places, [unintelligible]. Several attempts will be made to correct words and phrases of this nature. In cases where the unintelligible words or phrases are clearly not of importance to the meaning and understanding of the sentence, they may be omitted without marker insertion.

2. Transcript

My name’s Andrew Tridgell. I’m going to tell you a bit about rsync, and I’m going to be concentrating on the rsync algorithm for this talk, as opposed to to the rsync tool. A lot of people are familiar with the tool itself, but there’s actually some interesting technology underlying the tool, which I’m trying to get a few more people to know about (the technology), because it has some uses outside of the tool. And I want to, hopefully, see it used in a far wider range of applications than what it’s currently being used in. And so I'm hoping that by introducing a few more people to the ideas underlying the internals of the tool, that it might be implemented in a few more places. And I'm hoping that, eventually, the algorithmic part of the tool will become as ubiquitous as QSort or memcopy or other standard algorithms... standard utility functions that people use all the time in their programming. *[00m, 55s]*

So the core of rsync is this algorithm that I call the rsync algorithm. And it solves this problem, the remote update problem. Now the remote update problem is basically: you have two computers connected by a very high latency, very low bandwidth link... a typical Internet link, at least if you're in Australia. So, a piece of wet string, a really pathetic link... and you've got two files. The algorithm actually works on arbitrary lumps of data. You've got two lumps of data; one sitting on one of the computers and the other sitting on the other computer, and you want to update one of the lumps of data to be the same as the other one. So you're trying to just update a file remotely and you don't know... it's really like the morning after algorithm. It's an algorithm meant for: when you didn't have the forethought to, at one end or the other, have a copy of the old file so you could run a diff, or an Xdelta, or some other tool like that. So you don't have any way of doing a local diff between the two files and sending that. *[02m, 09s]*

The rsync algorithm is a way of solving this problem and much like neural networks, the last result for people who don't understand the problem, rsync is a good way of solving this problem when you don't know exactly what types of changes have been made to the data. There are always more efficient algorithms than rsync. If you have structured data, and you know precisely the sorts of updates, the constraints on the types of updates that can happen to the data, then you can always craft a better algorithm than rsync. And that's preferable by saying well you could always craft rsync, but in fact you'll very likely be able to craft a much better algorithm. But rsync is useful because it doesn't require any thinking; and it doesn't require any forethought to organize, to have particular information about the old file stored beforehand. So it's like this morning after algorithm for copying data when you need to update remote data, you don't know the old data, or you don't have any control over the remote changes. For example, you may not have any knowledge about what changes we may have made at the other end of the link because you're not the person who made those changes. And yet you're the person who has to update from that file. *[03m, 31s]*

Okay, so how do you go about doing this? Now prior to these sorts of algorithms, really there were only two ways of doing this; one was that you could just throw some compression at it. You could just compress the file and send the compressed file. That might gain you a factor of three, a factor of five, a factor of a hundred if it's a spam file or something. So you might be able to get some speed-up if the file is reasonably compressible. *[04m, 02s]*

But that really isn't enough, you know; people want more of a speed-up than that. Just because there's a lot more data there; there's a lot more information you're not taking advantage of. You have the old file. It's likely that the old file has a lot in common with the new file, because usually in a big file, the changes are small. Where I'm deliberately being vague here, small on an undefined scale. As soon as you start defining the scale of the size of the changes, actually defining those, you immediately have to get into the area of structured changes and structures to the file, in which case you probably shouldn't be using rsync anyway. *[04m, 43s]*

So the next thing is: you could keep the old file; you could have some forethought and arrange at one end or the other to always keep the old file and that way you can always send a diff. That's not always practical. So the remote update problem is: is there a better way of doing this, of updating the files remotely, without all of this prior knowledge, and doing better than just compression; taking advantage of the data in the old file? *[05m, 14s]*

First of all, maybe we should actually get you guys to give some ideas. Those of you that have already read the paper about rsync maybe shouldn't say what the idea is. Can anyone here suggest how you might do this; what are some... What I find is that, it's a bit embarrassing, really. I took several years to work on the rsync algorithm, but generally when I describe the problem to somebody, as a fresh problem, most people in the CS Department back at ANU worked it out in about half an hour. But then again, they knew there was a solution or I wouldn't have been asking them. So maybe with a few people, you'll work it out much quicker. Does anyone have any suggestions? How would you go about doing this? *[05m, 53s]*

[The speaker calls on an audience member.]

"Split the file into half and then try to do a checksum... keep breaking it up into smaller pieces until the checksum matches..."

Indeed you could; the problem is: what if someone has inserted one byte into the beginning of the file? Then neither half is going to match the half at the other end. That will only work if you've modified exactly half of the file or you've kept things aligned. If the alignments have changed at all, if any of the data has slid in the file, even by one byte, then any algorithm based on matching blocks with corresponding blocks in the other file won't work. Okay, so another idea... *[06m, 32s]*

[The speaker calls on an audience member.]

"You could take sixteen bytes and send that and then have the other end look for that in their file. Checksum over, say, a kilobyte after the match."

Okay, you could indeed. Now if you're sending just sixteen bytes at a time, you've got a lot of round trips. This is another one of the constraints. I'm going to change the problem on you. And I'm going to add the additional constraint that I want to do the whole thing in one round trip. In fact, I want to do it in one round trip for any number of files, even if you have ten-thousand files, I want one round trip in total. Just to make it a bit harder. Let's consider just one file at first, though. *[07m, 08s]*

There's still another slight problem. Say you sent checksums of every 16 bytes to the other end. Well first of all, that's going to be big, it's going to be a large percentage of the file, so 16 is too small. So lets make it, say, 600 instead. Okay, then the checksum is going to be 1% or something, of the file, something of that order. Which is getting more reasonable. *[07m, 32s]*

Then you've got a slight problem that you've got to checksum it at every offset at the other end. Now say it's a megabyte file at the other end. That means you'll be calculating a million checksums. *[07m, 44s]*

[The audience member continues.]

"Wait, I said just send 16 bytes, not the checksum of 16 bytes. So then you can just look for that 16 bytes..."

Right, and then you're getting very close. The very first version of rsync used something very similar to this. Rsync actually came out of some work I did on search algorithms. How many of you know about the Boyer-Moore-Gosper string searching algorithm? Very, very common. We have basically a delta table and you jump the length of the delta table. *[08m, 14s]*

I was doing some work on searching large text databases and I wanted to search for multiple alternates... We were looking up in the thesaurus; somebody was looking for something like apple and what you do is you look up in the thesaurus for all the synonyms of apple in different languages; pears and oranges and related words... and you want to search for any of those in a single pass, so you can do a thing called a multi-alternate BMG algorithm. And that is a very fast way of doing that. *[08m, 43s]*

The very first rsync algorithm did in fact do that. It sent chunks of data; what it did is divide the file into blocks and it sent the first few bytes and the last few bytes of each block and the checksum of the whole block. Then what it did is use this multi-alternate BMG search algorithm to search for the beginning of the block or the end of the block in the file. And where it found that the beginning of the block matched, it checked: does the end of the block also match and does the whole checksum of the block match? In which case it says: we've probably got a block match. *[09m, 13s]*

So in fact you're close; unfortunately it ends up being rather slow. It does work, though, and rsync 0.01 did that, but it was unacceptably slow. Because although multi-alternate search algorithms are much faster than standard BMG-type algorithms, they're still far too slow; we actually need a search algorithm that's linear and about the same speed as a memory copy. *[09m, 37s]*

Okay... other ideas...

"GNU diff takes the file and splits it into chunks after certain terms and checksums each line and then searches for checksums which match that line."

First of all, text based; not much good. The other thing is GNU diff relies on the bandwidth between the two files being very, very high. It's local. Both files are local. *[10m, 04s]*

"But it could be done in a shifted fashion..."

Not using the diff algorithm, no.

"How about not using the diff algorithm directly, then..."

This is what Xdelta is.

"... you can send the checksums..."

Yes, you can send checksums; what you'll end up with is basically something like the rsync algorithm; you're exactly right, but as soon as you have differing sized blocks, then you'll find that the bandwidth requirements to get sufficient data to the other end for the search algorithm will be massive. So those local differencing algorithms can use differing sized blocks, right, they can use carriage returns and things, but as soon as you go remote and you're worried about the cost of sending the signatures, and the cost of doing the search and things, you really have to go to even-sized. *[10m, 47s]*

Okay, other ideas? *[10m, 49s]*

"We combine the first suggestion of progressively doing like a binary breakup of the file with looking for the blocks that we've broken up and moving them."

So you could break it up into blocks on one end and on the other end, you could look for it at different boundaries and not just the block boundaries. Okay, that's getting closer. Yes, in fact you want to look on every boundary. You want to look at every single byte boundary at the other end. *[11m, 13s]*

"Send back offsets, as in we found this, but it was offset this far..."

Right, you could, yes. But the problem there is still... sending it back is no problem; the data format for sending it back... there's many, many formats you could use; it's more the fact that it's an N-squared... If you're doing your checksumming at every single offset, then you've got a million different offsets in a 1 megabyte file. You've got to do a million strong checksums; you're going to be there all day. It's too slow. *[11m, 51s]*

So how do we make that faster?

"How about a rolling checksum?"

Right. That's exactly right and that's the core of rsync. Basically, rsync combines a rolling checksum with a strong checksum. The rolling checksum is not critical to the algorithm except for speed. The rolling checksum is a filter on the strong checksum, to make sure that you only run the strong checksum when the rolling checksum matches. *[12m, 21s]*

Now for those of you who don't know what a rolling checksum is, I'll start explaining that. In fact, I'll explain that now, then get on with the rest of the talk. A rolling checksum is an update-able checksum or update-able hash. Say you've got a buffer here, you know in the middle of the air about there. And you know the checksum of that buffer and you want to know the checksum of this buffer, one byte along, where one byte has been added on the end and one byte has been removed from the beginning. So you've slid a little window along by one byte in a file. Now a rolling checksum is a checksum where you can calculate the new checksum from the old checksum with a very small number of operations. Maybe three or four adds and a couple of subtracts and a XOR, or something. A very small number of operations to get from a checksum in this location to a checksum in that location. And that's called the rolling property of a checksum. And that is the key to making rsync efficient. *[13m, 21s]*

Okay, so let's have a look now, to answer that question: yes there is a better way and the basic algorithm goes something like this, although those of you who have been following the little bit-windowing so far will now already know this and you can go and look at one of the other talks. [The audience laughs.] *[13m, 37s]*

First of all, we do signature generation, and what we do is we generate what I call a signature block for the old file. Okay, in fact you can invert this whole thing and you can send the differences in either direction. Either from the receiver to the sender or from the sender to the receiver. There's two different ways of doing it; I'm going to explain the one that is in the rsync tool. The one that is in the rsync library, which is part of some other applications of rsync, in fact does it everything in the other direction, for various reasons that I'll get into later. *[14m, 07s]*

Signature generation... we generate a signature block. Now the signature block consists of a checksum; or you take a file of say a megabyte and divide it into, say, six- or seven-hundred byte chunks or kilobyte chunks, say, you know, chunks of about that size. And you take two checksums on each block. One of them has this rolling property that you mentioned. And the other is a conventional hash. I use MD4, because I happen to have an MD4 routine laying around which is nice and fast. *[14m, 40s]*

So you take these two hashes on each block and you bundle them up and you put them all together, and that's your signature. The signature is just the checksums of each of the blocks concatenated together to be a lump. The overall size of this is about 1% of the file, of the order of magnitude... there's various maths you can do to work out the optimal size, depending on various assumptions which are never valid, but the rule of thumb is about 1%. *[15m, 11s]*

The next part of the algorithm is the signature search. So you do this signature generation and you send the signatures to the other end, to the guy who's got the new file. Right, so you do the signature generation on the old file, send those across. So you've used one half of your round trip already. So I said, rather meanly, to the first suggestion here, that I only wanted one round trip, right, adding a condition to the problem, which is the classic way of doing things. *[15m, 39s]*

So we send that and we've got one half of the round trip gone already. So we've only got one send left and then we're done. What we do then is that we now have to search for blocks from the old file that are present in the new file at any offset. And that's the trick. You have to find them at any offset at all, any byte offset, because that way you handle arbitrary insertions and deletions. Otherwise, you'd only handle block size. It'd be fine for VMS with record-based filesystems, but it'd be terrible for UNIX, where things tend to be byte-oriented. *[16m, 18s]*

Now that search algorithm uses this rolling hash and what it does is it takes the hash and rolls it along the new file, calculating the weak rolling hash checksum at each byte offset, looks it up in a little hash table and if the weak checksum matches, the rolling checksum matches, then it goes and calculates the strong checksum. If the strong checksum and the weak checksum match, then you assume you've got a block match and you've found a common block between the old and the new file. *[16m, 51s]*

Then you've got a simple encoding mechanism to encode that information back to the receiver, the guy with the old file. So we can reconstruct the new file from this encoded data. Now what does that look like? Let's have a look. So here's a bit of a diagram of how this turns out. Now have we got a little pointer here somewhere? No? I'll just use a finger. *[17m, 16s]*

What we've got is this is the old file sitting up there, file A, and that consists of these chunks of data. Now I've just drawn them with different hashes. So when, see this cross-hash goes this way, down to the left, assume that this is the same as this one which also goes down to the left. And this one here, which is a little brick-like, is the same lump of data as that brick-like one over there. That's what the little diagram is supposed to mean. It's all very obvious, I hope. *[17m, 55s]*

Okay, so here's our new file and here's our old file and this is the resulting difference, which gets sent over the wire. And the resulting difference, the first block of the new file doesn't exist in the old file, so we have to send it as literal data. We have to send that over the wire, right, because that data does not exist in the old file, therefore the information is not present on the receiving computer. You've got to get it there somehow. It's basically got to go over the wire. *[18m, 30s]*

There are second order things you can do, you can do things like compression on this, no problem. Rsync does that if you use the -z option, it'll compress that. And there's some tricks you can do to make that compression go especially well and things. But basically you've got to send it. *[18m, 45s]*

But this second lump of data in the new file is the same as this lump over here, which is just in a different position in the old file. Okay, so the search algorithm will find a match between this little block here and this block here, and will emit a token. The token is just an index into the indexing count or whatever you want to encode it as, it's a RLL-type encoded, RLE-type encoded in rsync, token which represents the match from the old file, the location, which is saying to the destination: when you're reconstructing this new file, shove out this little literal block of data, then go and fetch token number 0123 from the old file and shove that out. *[19m, 39s]*

Then we see, the next lump of data, this hash here, is the same as that hash over there. So we send another token. Then there's another token and and this lump of data is another lump of literal data, that's that lump of literal data which doesn't exist anywhere in the old file. So what we end up with is; the difference going over the wire is alternately lumps of literal data and tokens representing lumps of data from the old file. *[20m, 04s]*

Okay, so that's our difference. And we can generate that from the searching for the signature blocks in the file, and that's all fine. So it's not really that magic when it comes down to it, which is why people work it out in a few minutes, and why it took me several years. But it's very easy to code; you can code up the rsync algorithm in a matter of a few hundred lines of code. You can make it fast in not much more than that and so you should be using it in all of your programs. *[20m, 40s]*

Okay, so let's go into it a little bit more deeply: signature generation. The file is divided into uniform blocks on the receiving end. Now it has to be uniform blocks if you want to make this thing at all efficient. I'll let you think about that yourselves. I know people here are already thinking: how can I do it in non-uniform blocks, optimize it? There are ways of doing non-uniform blocks, but only if you're willing to handle more than one pass. Or you're willing to increase the amount of signature data that gets sent over the wire considerably. *[21m, 10s]*

So, because of latency, those of you that have done that much network programming will know that latency is at least as important as bandwidth, if not more important. Particularly if you're in Australia. And bandwidth really can be quite cheap but latency is often a killer, and so I really wanted to keep the latency to an absolute minimum. So I wanted to keep it down to one round trip total. Of course TCP throws extra, you know with its windowing, it's ACKs and all that, but at least because of the windowing and the smarts in TCP, the costs of the extra round trips and things aren't really that high. Whereas logical round trips up at the algorithm level would be a killer. *[21m, 50s]*

A good demonstration of that is: if you look in I think chapter three of my thesis, I've got a little table where I show the speed of rsync-ing a thousand small files, like one byte files, versus FTP, versus RCP. And rsync was like two seconds and FTP, (this is over a modem) and RCP took up like the order of minutes. It's an enormous difference. And that wasn't the rsync algorithm doing anything, because it can't do anything on a two-byte or one-byte file, it was the fact that it was saving latency; the fact that it was using one round trip for all thousand files, rather than doing one round trip per file. So a lot of the time when people find rsync fast, it's not because of the algorithm underneath it, it's because it saves latency. *[22m, 35s]*

Okay, anyway, it's divided into uniform blocks. On each block we calculate two checksums, one of which has this special rolling property. Additionally, we also calculate a whole-file strong checksum. Why do we do that? To make sure we reconstruct the file properly. Do we actually need that? You could have a bad link, but we're already doing checksums. The first version of rsync didn't have this and yet I can assure you that the first version of rsync will not fail, it won't fail in the lifetime of this universe anyway, except for bugs, there were plenty of those, but it won't fail due to the algorithm. Why not? *[23m, 22s]*

[There is an unintelligible response from the audience.]

Yeah, that they're the same to begin with, the algorithm would be very, very efficient. The difference would be just a bunch of tokens and then run-length encode those and it'd be like four bytes. So that's not a problem. *[23m, 36s]*

The reason the whole-file strong checksum is needed is so you can cheat on the block checksums. We can use really short block checksums. If you use something like a full MD4 for each block, 128 bits per block, then there would be no point at all in having the whole-file checksum. Because you're already doing a 128-bit checksum per block. Right, but 128 bits per block, 16 bytes per block, is very expensive. That's a lot of checksum data. It means you'd have to have large blocks in order so your checksums don't swamp the link. So instead, what you do is you trim down the size of the checksum per block; you make it really small. Which means there's a probability, there's a small probability that the algorithm will fail. You have to catch that. *[24m, 30s]*

And you catch that with the whole-file checksum which you can do at the same time; you can calculate at the same time with nice caching properties, etc. as you're doing the individual block checksums. And that way you know when you've screwed up. *[24m, 46s]*

In fact, it turns out that it's extremely rare this is needed anyway, because the rolling checksum is actually quite strong. It doesn't need to be that strong, but it just turned out that the algorithm I used, which is a derivative of the Adler checksum, is... it's a 32-bit checksum, two 16-bit halves. It has an effective strength of about 28 bits, which isn't bad for a rolling checksum. There are better ones, though. Somebody at defense in Australia has one with an effective strength of about 31.5 out of 32, but it's a lot more expensive to calculate. *[25m, 19s]*

Anyway, so you need this so you can use a small block checksum. The current versions of rsync use a 16-bit block checksum, plus a 32-bit rolling checksum, giving you a total of 48; effective strength of about 46. Which is still plenty so that none of you, if you're running rsync all the time have ever noticed a resend and using the strong checksum, but it's there because it will happen once every, you know, hundred years or so. So you do have to make sure and catch it when it does. *[25m, 53s]*

Okay, the signatures are sent to the sending side, obviously, and the optimal size depends on the data. Although the dependency is a fairly flat one. If you look at a graph of optimal size of the signature size versus the total data transmitted, the efficiency of it, it's a relatively flat graph. So you can play with it on the command line of rsync with the -b option; rsync has got "option-itis" quite badly. You can play with that if you want to, but you'll find that it's a relatively flat graph except in some particular types of data, when playing with that can give you massive improvements. *[25m, 34s]*

"Why do you use MD4; why not use something simple like CRC16?"

Primarily as a PR stunt. When rsync came out, nobody trusted it. Everyone said: it can't work; this is going to fail all the time; the probability for failure is blah and blah. And there were all these people saying it's not going to work. So me being able to tell them that rsync failing is equivalent to breaking MD4, gives them a nice, warm feeling in the tummy. The fact is that no one is attacking the transfer, right. Because they're not attacking the transfer, a CRC is ideal. And in fact I would have used a CRC if it wasn't for the fact that users wouldn't have trusted it. *[27m, 15s]*

But you're right, from a mathematical point of view, a CRC would be far more appropriate, but MD4 is actually very fast. It's quite a bit faster than MD5; I already had one laying around that I'd written for Samba, because you need it for the encryption stuff in Samba; and it also meant that I felt very confident that I was transferring the right data. That was particularly when I wasn't using the whole-file checksum. I could now, for example, go to a CRC per block and go to an MD4 for the whole-file checksum and that would be fine, because then I can still say it's equivalent to breaking MD4, but why bother? That is not the bottleneck and in fact the bottleneck, when people use the -z option, 90% of the CPU is in gzip, you know, the zlib library. *[28m, 02s]*

[The speaker calls on an audience member for a question.]

"You talked about the possibility of failure being maybe one in a hundred years..."

Much, much higher than that. That is if... the probability of it even needing the strong checksum. The probability of failure is about one in 10-to-the-11th power years if you... I've got a slide on that, but basically this universe will be long dead before the algorithm fails. *[28m, 26s]*

[The audience member continues.]

"How did you test that? How do you know the resend code works?"

Oh, I know, exactly... well he's right, you need to test... code that isn't tested is broken. Everyone knows that. If you haven't tested your code, then it doesn't work. So what I did was I dropped it back to a 2-bit weak checksum and I tried like 2-bit and 4-bit and 5-bit and various ones which meant the probability of the resend was much, much higher. And I got it to resend every hour or so and that way I was able to test that the resend code does in fact work. In fact I even did one of the releases of rsync: accidentally had a one-byte, I think, because I'd left the constant wrong from some experimentation. It didn't matter, though, you know, it's just... anyhow. Yes, so maybe it actually got tested by someone else, apart from me. Which is good, yes, deliberate, of course. *[29m, 15s]*

So that's basically how it works and a failure probability, yes, you can work out... There was a rather amusing e-mail I got from one particular person worried about his data. He said: no, no, we can't do this, the probability of failure is huge; it'll happen all the time. And he had this great, long page of math; he said: if we assume this sort of data and blah, blah, blah, math, math, math, math... long page and at the bottom, it said it will fail at approximately 10-to-the-49th power years. Oh, I suppose that's okay. [The audience laughs.] I'm glad he still hit send, because that was quite cute. He was actually off, his math was a bit off, but it's a massive number, basically. The universe is thought to be about 10-to-the-10th power years old and if you assume one million transfers per second of a one megabyte file, constantly, then you end up with about one in 10-to-the-11th power years, approximately. So it's a big number. *[30s, 12s]*

Yes, there's a question over here?

"Yeah, I was just wondering... when you see a failure in one of the individual checks, what exactly do you do to recover it? Just send the whole file or..."

No, what I do is resend with the full 128-bit per block and I seed the checksum differently each round, so I use a different seed for the checksum algorithm. What I do is: with MD4, I put a seed at the front of the MD4 and MD4 hashes the data plus the extra seed at the front. Even if there was some particular failure mode of MD4 in the block algorithm, it's going to be a different seed every time and the only way you can tell is you see the file name twice. *[30m, 55s]*

But the filename twice can happen under other circumstances; if you've seen this happen, it's almost certainly because the file changed during transfer. Rsync does no locking. Which means that: if you are modifying a file while it's being transferred, then probably the checksum will fail and it'll go round again. And if it goes around twice, and it still fails, then it prints a message saying; Error, checksum failed, file changed during transfer? And it's probably a file like a log file that's being constantly updated and so the checksums didn't match because it's never going to be able to get it exact; it means that what you've got on the other end is something which will approximate some snapshot of the file, but because it's not doing any locking, it can't guarantee that it's got a particular snapshot of the file, because you can't have an atomic read of the whole file. *[31m, 49s]*

"Why is there no locking?"

Because nobody's submitted a patch for locking yet. It doesn't do locking for a number of reasons. Locking is cooperative in UNIX and has to cooperate with some other locking scheme... which locking scheme? I could flock() it, I could do something like that. What do I do if it's locked? Do I then block the whole transfer? And then people use this in cron, etc. What I'm thinking of doing for a future version is having a locking option, –locking=flock, –locking=mailbox, –locking=something, you know, various schemes and you can drop in different locking schemes if you want to. If you're silly enough to use mandatory locks, you could actually make it work properly. UNIX doesn't really use much locking and so UNIX tools don't tend to lock stuff. cp doesn't lock and other tools don't lock, but you're right that it could be nice in some cases to do locking, particularly if you're transferring known types of data when you know the locking-style of that data, like mailboxes. *[32m, 54s]*

What I tend to do for mailboxes, in fact all of my mail gets delivered and sent using rsync, I've got an rsync-based mailer; it's a tiny little shell script. It's in /pub/tridge/misc on samba.org... about a couple hundred lines of shell and it does send/receive SSH transferral. It's actually nice stuff, a nice little mailer. Anyway, plug number... And it uses that, what it does is inside the rsync, it calls the movemail and that of course does the right locking stuff, atomic and... you know, wondrous things. Okay, that's that. *[33m, 36s]*

Some of you may not realize, you can actually do a lot of things with the rsync option -i, as I mentioned earlier. One of the cute things you can do... people get asking about things like: please can you add an option to rsync to run this command on completion? Well, no, because you can do that already. There's the rsync path option which says where to find rsync on the remote end. Now the thing on the remote end only has to look like rsync, it doesn't actually have to be rsync. So it can be a shell-script wrapper that runs rsync in the middle, checks the error code and does various things at the end. And that's how my mailer works. Therefore you can run arbitrary commands on (the completion of rsync) the successful completion of rsyncs, by using the rsync path option to specify a program other than rsync that happens to call rsync and send it to standard in/standard out at the other end. So anyway, those of you that like playing with shell scripts might like that. *[34m, 38s]*

The actual rsync program has a bunch of other features that make it a useful tool, which are completely orthogonal to the algorithm. Particular things... latency-saving design, I spent eight years studying from the other end of a modem from where all my data was, and so I tended to worry about latency and often my stuff was on the other side of the world as well, so I really cared about latency; it really cares a lot about latency. Fancy file list handling, excludes, skipping files, that sort of thing. *[35m, 11s]*

SSH support, people seem to like that a lot, it's very useful at getting you secure transfers. And thank god OpenSSH now has finally got non-blocking I/O. That was a major pain with rsync, the blocking I/O and various bugs with pipes in kernels of various operating systems and things. Finally, OpenSSH 2.1 and above does non-blocking I/O which means rsync works much better. So if any of you are using rsync over SSH, then grab the latest OpenSSH; things work a lot better, a lot more smoothly. *[35m, 43s]*

Daemon mode... rsync is now becoming quite a common distribution tool for various places. A lot of sites that have FTP sites also have rsync sites. And it sort of exports shares and that sort of thing. *[35m, 47s]*

Incremental backup is one that people aren't aware of. Rsync actually can be used as an incremental backup tool. Including doing like rotating backups and all sorts of things like that, using the –backup-dir option. I should put up some more examples of that on the rsync site. We use it for all of our backups; we have a backup server in the office and rsync has built-in the capability to say: okay, update this remote directory as a backup, but any of the changed files, deleted files, put them in this incremental directory. And that way you can have a seven day rotating backup and that sort of thing, which is quite useful. But a lot of people aren't aware of that. *[36m, 35s]*

And in all of this stuff, the significance of the actual underlying signature algorithm, search algorithm, really has been lost. (Now, don't know how much power I've got left in this; don't want it to embarrassingly turn off in the middle of the talk. Oh boy! Power. Where's the plug? I'm about to lose... I'd better suspend. Great. Right, saved! I hope. Battery charging... yes. Zero percent. We just made it. Okay, that was close. There was this light flashing; I wondered what it meant. [The audience laughs.] I just got this laptop...) *[37m, 31s]*

Right, other uses for the algorithm. Rsync is starting to get used in a few other places, which is really good. I'm hoping that it's going to be used in a lot more places, though. Perhaps the most important one is rproxy, which is embedding rsync into the HTTP algorithm. And I'll talk about that in a second. Then there's also an effort under way to embed rsync into CVS. Rsync as part of a WAN filesystem is one of my favorites. And rzip is something I developed just out of academic interest; it's a very good compressor. It does much better compression, particularly for large files, than anything else I've seen. But it's extraordinarily slow; really, really slow. But if you really need that extra compression and you've got a few weeks to spare, then you know, it can be good... hours, anyway. *[38m, 26s]*

So lets talk about some of these a little bit to give you some idea of the sort of things you can do with this. Rsync in HTTP is one of the most important ones. And that's nearing the point where it's ready for release. It's a thing called rproxy. And it's being developed by another guy at Linuxcare in Canberra [Australia]. And what this does is basically, think of the history of the Internet, the web, you know, of course the Internet is the web. Think of the history of the web. Basically, we started off uncached; everyone just had a browser, so you know the old browsers. Then caches were invented because it made things much faster. And then CGI came along and everything got slow again. Basically, one of the big problems is there's massive cache infrastructure worldwide. Right, people have Squids everywhere and hierarchies of Squids and NetApp caches and all these sort of things. The problem is that they're all completely useless for dynamic content. And yet the world is moving towards dynamic content at a very fast rate. *[39m, 26s]*

Probably by volume of data transferred, the majority of the data on the web is already dynamic. Or at least it's getting that way. So how do we solve this? Well what we can do is you can actually, there's a couple of proposals to solve this; some people are proposing to, in the markup, in the HTML or XML itself, you markup sections that are static and sections that are dynamic, etc. The problem is web site designers are inherently lazy, or brain dead or something... something about them. If we can't even convince website designers not to put four megabyte images on their home pages, how are we going to convince them to put in the markup sections for dynamic and non-dynamic and get them right? It's too much effort. We really have to do it in the tools, so it just happens. I don't think those markup things are really a practical solution, for the wide range, really big use. *[40m, 23s]*

The other types of solutions... there's a solution being proposed at the moment, HTTP delta. Unfortunately, it really relies on storing outgoing pages. Having some sort of database or repository where you store all pages that you send, so that you run a diff, then they send a tag ID to say which page they've got in their cache, and then you can generate a difference. The thing is, I run a fairly large website myself and I just don't like the idea of every page I send, I have to store on local disk. Because I send a lot of pages. And you know, storing all of those pages would be a great pain. *[41m, 00s]*

So what rsync and HTTP does is it solves the problem by allowing only the differences to go over the wire, without having to store the old files. Using this exact same algorithm. So what you do... it has all of the nice features... it builds on the existing web infrastructure. You know, sounds like a sales pitch. All of the content is cacheable, whether it's CGI, whatever. Any content becomes cacheable and there's no extra round trips. Because extra round trips would be deadly. We've got enough of them already with TCP handshakes and all that. *[41m, 32s]*

So what you can do with rsync and HTTP is... imagine the client has a cached file. Let's assume we haven't got a chicken and egg, we've already got a cached file there; we've cached some CGI file, okay. The client generates the signature from that cached file and adds it to the request as an rsync signature header. In the request for the... re-requesting the same page, it adds this header; it's BASE64 encoded or whatever. Then the server generates the page as usual, so the server runs its usual CGI generation stuff, then just before it goes on the wire, it says: oh, did the incoming request have this rsync signature header? Oh, it did, then I can generate a diff. Because it can use that signature plus the new page to generate the diff from the old page. Because it doesn't need access to the old page to generate the diff. The signature will do the job. *[42m, 34s]*

So it generates the page as usual, does the checksum search, and generates the rsync differences, then it sends the rsync differences as content-encoded rsync and (decodes it to give the new page,) sends it back to the client and the client decodes it and shoves the new page in its cache. And this actually works, it works very nicely and I get massive speed-ups; I've been running this for a little while at home, across my modem link. I run it for the last hop over the modem. Now I've got ISDN, so I haven't bothered running it for a little while, but it reduced the traffic over my modem link for web surfing in general over a two-week period by 90%, which is a fairly large reduction. *[43m, 15s]*

The fact that I throw gzip over the top of it as well helped, you know, it got some of that. But a fair bit of it was in fact the improvements from the algorithm. The rproxy project is a project to do this properly; I just did a rough prototype. And Martin Pool, who's been working on Apache for quite a while, is now working on building this into Apache and doing it as a separate proxy plus possibly doing patches for Mozilla. Though we haven't started looking into that yet. And also doing patches for SQUID. And the idea is that eventually, you have it in your SQUID server, you have it in your web server, and then you end up just sending differences for CGI pages between the web server and your local SQUID. *[44m, 03s]*

The advantages of doing that, of course, is that if your browser doesn't understand this extension, you still get the benefit. Because the way we've organized it is that you get the benefit between the two furthest points in the HTTP chain between the client and the server that understand the extension. So if the server and your SQUID both understand it, you get the advantage; if the second-level SQUID and the first-level SQUID understand it, but the server and client don't, you get the advantage for the hop between the two Squids. So if you've got some slow link or something, you can put one of these over it and reduce the data going over that section. *[44m, 40s]*

[There is a comment from an audience member.]

"It seems to me that this really only benefits dynamic content, it seems to me that it benefits transcontinental, transpacific proxy caching efforts, but really doesn't do anything at all for the... in fact, it doesn't even for pages that are static or mostly static and change weekly or something like that. But it doesn't really even... My impression is that the bottleneck for most of those dynamic content is that those CGIs are just too slow and they just..." *[45m, 15s]*

No, no, no. If you have a look at something like... if you go to something like CNN.com, the page remains exactly the same every time except for about 8 bytes. They have this little ID that ticks over on every page, okay. Which means that with current technology, you're resending that 150KB every time. Now the generation of it... you can solve the generation problem by just getting a faster CPU. But it's relatively cheap to get a faster CPU, or just get more of them, have a farm. But it's harder to get a faster network link. What this does is it allows you to trade off between CPU speed and network speed; this costs you some CPU speed... it's actually quite fast, you can saturate a 10 Mbps network quite happily; with a fast enough CPU, you could saturate 100 Mbps. But if you've got a bit of CPU around to spare but you haven't got a particularly fast network link, then you can use some of that CPU to reduce the amount of bandwidth you need. *[46m, 08s]*

"Do you see it over a 28.8Kbps modem, or any high latency link?"

Come and live in Australia for a while. [The audience laughs.] It's not for everyone, you don't want to use it on an intranet, it's just going to be a waste of CPU on your intranet. But there's an awful lot of places... I've heard that a lot of people in the US have DSL; something like 30% of households have access to DSL, or could have DSL if they wanted to in the US. It's like 0% or close to it in Australia. But the other 70% could benefit from this sort of thing. You know, getting that last mile to the end-user. This could actually benefit quite a lot. Okay, so, and of course you don't have to turn it on, it's just a feature. *[46m, 56s]*

Rsync and CVS... Ben Elliston is working on that. Hopefully, it's going to be done any day now for the past several months. It's that kind of project. *[47m, 05s]*

Rsync and a WAN filesystem, this is quite interesting. You can combine rsync with lease-based filesystems to do the... You combine rsync with a WAN-based filesystem. Now lease filesystems, it's a bit like the stuff in NFSv4 where basically the client can take a lease out on the file and do write caching and read caching on the file. *[47m, 39s]*

Now imagine you had a WAN filesystem, a global filesystem, where you're reading your mail from a machine in the US and you're sitting in Australia. And you load your 10MB mailbox, you subscribe to Linux Kernel [Mailing List], so you've got this massive mailbox and you bring it up, then you delete one message out of it, then you save the mailbox again. *[48m, 07s]*

Now with normal network filesystems, what you would have to do is send that whole 10MB file, chug, chug, chug, chug, chug, back to the server, because you've written a modified file. The length has changed, everything has changed about it. Right, but if you've got a lease on the file, then what will happen is those writes will initially go into the page cache on the client. So in the page cache on the client, you have the new file. In the page cache on the server, you will have the old file. So you have two lumps of data and a low-bandwidth, high-latency link between them. What do you do? You synchronize the page caches between the client and the server using the rsync algorithm. What that means is that what goes over the wire is just a few bytes, because it says: oh, all of this part of the file is the same, and all of that's the same, and there's these few bytes that are missing. *[48m, 55s]*

Notice one of the things, I didn't point this out earlier, about the rsync algorithm that are very different to diff. Diff only handles insertions and deletions. Have any of you noticed that if you move a routine in a file, then you send the diff, what it does is says minus, minus, minus this one and plus, plus, plus that one; if you use ed-style diffs it just says edit out this bit and add that bit. It repeats the whole chunk in its new location. That's because it can only handle inserts and deletes. It encodes everything in terms of inserts and deletes. Whereas if what you actually have is a move of a lump of data in the file, you can't encode that with a diff-style algorithm. Rsync encodes that as a move. Because it just says: this token is at that location. So you can handle arbitrary moves. Which is particularly good for remote filesystems if you've sorted the file or done something weird to it, you've moved a chunk of data around and resave the file, it will encode that. It also handles binary files quite well. The thing about gzipped files, if I've got a second, I can tell you about those... Question? *[50m, 02s]*

[The speaker calls on an audience member for a question.]

"The brand new filesystem there..."

It doesn't exist yet...

"but in theory, though, there's a significant problem. You're saying updates to the file are maintained in the page cache and I presume the rsync algorithm updates to the fileserver occur when the file is closed..."

Or when it's flushed. Typically with lease-based filesystems, you flush every 30 seconds or minute or whatever. *[50m, 25s]*

“Okay, so even if the file is still open...”

You’re worried about turning off the power and losing it...

“No, I was more thinking if you make so many updates between fopen() and fclose() that you fill up your page cache...”

Hang on. But if the data is in memory, right... now it doesn’t record the changes you’ve made. It’s not like... there are other algorithms; there are other systems and what they do is they record all the edits, then they send the edits as an edit list, which can become infinitely large if you made a large number of edits. That’s not how this works; it’s when you save the file you’ve got the whole file as the new file and you’ve got the old file at the other end. You don’t have a list of edits available to you. To have the list of edits, you need to instrument all of your applications; you don’t want to instrument your applications. So you’ve got this lump of data on one end, the new file, you’ve got the old file as a lump of data on the other and all you’re doing is using this as a fast way of getting the old data to equal the new data. *[51m, 26s]*

[The speaker calls on an audience member for a question.]

“So it sounds like an interesting application would be rsync-diff.”

The thing is that contexts are a problem. You can’t get contexts. *[51m, 39s]*

[There is an unintelligible comment from the audience member.]

That’s Xdelta. Have you seen Xdelta? Xdelta is an rsync-inspired delta algorithm; it’s a very nice delta algorithm. It works really well on binaries. It works really well... and produces very small differences. It was originally based on the rsync algorithm. The thing is, since it’s a local differencing algorithm, he’s got high bandwidth between the old and the new file. So he was able to tweak it and tweak it and tweak it and tweak it and now really it’s nothing like the rsync algorithm. It’s really just inspired by it. And if you want a local differencing algorithm, use Xdelta. It’s a hell of a lot better than using a local rsync because he knows that both are local, he can do all sorts of tricks to make it really fast. In fact, it will give approximately the same differences as if you had run rsync with the -b4, block size of 4 bytes. *[52m, 31s]*

[The same audience member asks another question.]

"Well what if I were doing a cvs diff from a remote repository?"

Ahh, well that's where we are building it into CVS. Inside CVS there is a file send operation and the file send operation on the other end you have the best guess of the old file. It doesn't have to be the old file; it just has to be your best guess at it; the last revision or whatever. Then what you do is inside the file send method, inside CVS, you go file send rsync method; you don't call out to rsync, you just call the library and that's what Ben's adding to CVS. *[53m, 09s]*

"I'm surprised the client doesn't diff locally in local mode."

It's got both files then and runs diff.

Yeah, and that's what Ben's adding. He's adding it... it helps annotate, it helps commit, it helps diff, it helps anywhere inside CVS, if you do a cvs -t (trace), when you see sending file blah. Those bits get sped up. *[53m, 31s]*

[There is a question from an audience member.]

"Any special issues in copying directories to directories?"

Well the rsync tool has all sorts of options; does recursive and knows about symlinks and directories and devices and inodes and rsync is quite commonly used... In fact, I did an install when I changed hard disks on my laptop, from one hard disk to the other, with just rsync from / to / and then boot, I just had to run LILO. So it will handle all of the different properties. It will do devices and all those sort of things. And it knows about directory hierarchies and hard links and all that sort of stuff. The hard link support is optional; you have to add -H, and it's a bit slow; the algorithm for doing hard links is a bit horrible. But it knows all about directories and things. *[54m, 24s]*

"Is all of that stuff obvious when you think it through, or..."

No, there are some tricks to it. The main thing is latency-saving tricks. If you've got a whole tree of a million files, you don't want a million round trips. You don't even want one round trip per directory. So what it does is, in fact, one logical send for the whole tree. One logical send of a stream of signatures and in fact there's three processes; it forms a little triangle of processes. *[54m, 48s]*

We've got, at one end of the link, the generator and the receiver; and the generator is generating signatures on the tree as fast as it can and stuffing it down the link. The sender at the other end is receiving those signatures and is generating the differences and is sending it back to the receiver at the other end. The receiver is having differences thrown at it as fast as it can and generating new files. *[55m, 07s]*

Which means the whole thing is one logical trip pipelined for the whole tree. Which is why rsync works really well for large trees. Because of latency saving with small files. Also, the way it encodes the file list, transferring the file names, it has various tricks. It ends up encoding them as approximately ten bytes per file, including all of the various size information and things. Like it has a bit to say that this is the same device number as the previous file. It has a byte which says how many bytes are in common with this filename and the previous filename. *[55m, 41s]*

That sort of thing, so it packs it and compresses and this sort of thing. So there's various ticks in there. I went a bit mad, actually, and tried to save every bit. And it was a mistake. Because several people have asked about making this an IETF standard and doing some sort of RFC and that sort of thing. The format on the wire is too horrible to put in a document. Because I tried to save every bit. When you save every bit, you end up with a very specialized format and it was like rebelling against the SMB protocol. You know, I was doing something completely different. But instead what I'd like to see, if we're going to make anything a standard, I'd like to see a new tool developed on top of librsync which is much, much cleaner, but uses a sensible encoding rather than this saving every bit encoding. *[56m, 27s]*

[The speaker calls on an audience member for a question.]

"Compressed data, stream compressed and block compressed data; how does rsync work on those files?"

Okay, compressed data. Basically the rsync program works fine on compressed files, the actual binary works fine, but the rsync algorithm is not very efficient on compressed files. And the reason for that... it depends on the type of compression, but most compression algorithms... if you change the first byte of the file, then every byte of the compressed file beyond that point gets changed. Now there's an easy workaround. (You could just send the whole file.) *[57m, 07s]*

In fact there's two workarounds, one that everyone tends to suggest, is why don't you uncompress the file, send the compressed differences of the uncompressed file then re-compress at the other end. And the reason I don't do that is: sure you will get the right uncompressed data at the other end, but you won't end up with an identical compressed file because you won't necessarily have the exact same revision of zlib at both ends. A different version of zlib will decompress the file with no problem, but it'll compress it differently. It'll have different options; it won't necessarily produce the same compressed stream. Which means sure you will transfer the correct compressed data, but the MD5 checksum of the two files may not be the same. And that is a problem, because if people started using rsync to distribute RPMs, and they come up with a different MD5 checksum or tarred gz after the transfer, everyone would lose faith in rsync. *[58m, 02s]*

So there's a way you can get around this, but it requires a small tweak to gzip or bzip or whatever the compression algorithm is. And the gzip one is easiest to explain. Gzip has a 32K buffer, a history buffer. Now gzip uses dynamic Huffman encoding, which means if you change one byte in the file, everything after that point in the file changes in the compressed data. Problem; that means of course that rsync will be terrible, unless, of course, the change is toward the end of the file. *[58m, 31s]*

But what you can do is inside the gzip algorithm, you keep a rolling hash... (We've got an earthquake, have we? [The chandelier in the ceiling is shaking.] A truck, okay.) Inside the gzip algorithm, you maintain a rolling hash of the last say hundred bytes or three hundred bytes, or whatever number of kilobytes. In fact, ten kilobytes would probably be optimal. So the last ten kilobytes rolling hash and you roll that along while you're doing the gzip. Now whenever that rolling hash equals a particular number, and zero is a good number, then you reset the internal compression tables of gzip. *[59m, 11s]*

What that means is that the differences don't propagate more than ten kilobytes on average. So what you end up with... that's if you're using a ten-bit rolling hash... a ten-bit rolling hash would mean it's one kilobyte, but whatever... twelve bit, eleven bit, whatever. So it's a tiny tweak to the gzip algorithm and is one I've been meaning to submit to the gzip maintainers and the zlib people to basically make gzip rsync-friendly. And it doesn't affect the compression ratio much at all. Very, very small difference in the compression ratio. But it does mean that you've got a fifty megabyte .tar.gz file and you've got a different version of that .tar.gz file, then rsync will do very, very well on that. And it will end up with a very small transfer. So that's a proposed change. *[60m, 06s]*

In the case of bzip, you do the same thing, but with a longer... what you do in bzip is you select a block size; normally you select on the command-line. Normally, it's 900K. In fact, -9 means 900K, -n means n times 100K. What you do is you set the... you keep a rolling hash of about the length of that block size and you basically... you restart the blocks; you have to do a larger block in that case, so it would only be useful for really big files. Do a larger block and you reset the bzip tables when the rolling hash equals zero in each case. And you don't... when you do that, you don't keep your dynamic Huffman tables between blocks. *[60m, 52s]*

[The speaker calls on an audience member for a question.]

"You talked earlier about how you were doing the hierarchy, you treat it as one big thing; does this mean you can actually cope with when a file gets renamed?"

Coping with files that are renamed is partly in there; it's called the fuzzy option, –fuzzy. It's in the CVS tree partly; it's broken, it doesn't work yet. I started coding it a few weeks ago. The idea is to cope with renames so... it's for Debian trees and that sort of thing. I should call it the –debian option. [The audience laughs.] Where basically, you put out your .deb files or your source files or whatever and the filename changes every time you put it out and so what it does... I've got various heuristics for finding the closest match filename to use for the signatures. And it looks first in the current directory for files with at least six characters in common and various other stupid heuristics. And the idea is it will find files that are similar named that have been renamed. *[61m, 51s]*

If you want to actually handle arbitrary renames, it's very expensive because you need to be able to search for the signatures across all of the directories. It's an N-squared problem... and it's expensive. But handling renames to similarly named files is easy or doing on the command-line, if you have an option to say this file equals that file. In fact you can handle renames now if you want to do it all on the command-line, you can just say rsync this file to that filename. And it's perfectly happy to have different filenames on either end. Is that what you were asking about? *[62m, 23s]*

[The audience member continues.]

“No. I must have misunderstood how you said it works, because I thought you treated it as if it were one large file...”

No, I don’t treat it as one large file; it’s just that I pipeline the signatures from each of the individual files. *[62m, 36s]*

[The speaker calls on another audience member for a question.]

“What if when you transferred files reversibly, you made a tar file of the directory and transferred that?”

You could indeed do that, but it wouldn’t give you the nice properties. It would take an awful lot of disk space at the other end to store the tar file then unpack it. The way rsync works at the moment, the destination file is created as a temporary file, a dot file, then it’s renamed atomically at the end over the file. You can rsync tar files. And in fact in my thesis, that’s what I did. If you look in my thesis, all of my results are for rsync-ing of tar files. And the reason for that is that all I was interested in was the algorithm, not all of the associated garbage of directory handling and all that. So to get rid of all that, I just transferred one file. But I wanted it to be a realistic file, so I did tar files of the Linux kernel and emacs and things like that. So in fact, doing tar files is perfectly sensible. Although don’t make the block size 512-bytes; you get various things with tar files at 512-byte offsets. *[63m, 35s]*

[The speaker calls on an audience member for a question.]

“Another interesting thing you can do in the rename situation is you can use the inode numbers.”

Right, yes. Indeed I could do that; I hadn’t thought of that. *[63m, 48s]*

“You could look first at this magic inode table...”

Yes, yes. But the, hang on... Which inode number are you going to match, because you don’t have the old inode if you’re doing it between two machines.

“The master file on that side.”

But if you’ve got the master file on the same... you would have had to have cached the... you would have had to have done an ls -li previously and stored that somewhere. You don’t have the old inode number. *[64m, 20s]*

"You had to scan the tree anyway and look at all of these files."

You have to scan the tree on one machine; you have the new scan tree on both machines. You don't have the old scan tree on the destination. Which means you don't have the old inode number.

"You would have to be fuzzy on the opposite side..."

You would have to have some forethought and the administrator would have had to do an ls -li and stored the inode numbers somewhere. *[64m, 45s]*

"You could extend the rsync protocol to send them over..."

I do send inode numbers when I'm doing hard links. If you use the -H option, it sends device and inode numbers, because that's how it detects hard links. *[64m, 58s]*

[There is an unintelligible comment from an audience member.]

I'm not... maybe we should talk about it after... I'm not quite sure that works. *[65m, 08s]*

[The speaker calls on another audience member for a question.]

"How well does the rsync algorithm work for really huge files? Does it still use about 1% of the file size?"

What you do is... the optimal... it works fine for big files. Optimally you should scale the block size as the square root of the file size. In practice, I scale it linearly with the file size. And what happens is as you get a really, really big file, you get a really, really big block size which means you don't see small matches between the two files. And the reason I scale it like that is because I like the fact that rsync with its default options will never be worse than 1% over sending the whole file, okay, because the signatures are approximately 1% of the data length. Whereas if you scale it according to the mathematically optimal, which is approximately the square root of file length, then the problem is that the worst case becomes quite bad in the case of large files. *[66m, 08s]*

You can end up with quite a large percentage over the case of just sending the file. But you can tune that on the command-line. There are some cases where people are doing very large DNS, you know, millions of machines type stuff zone files and they're doing lots of updates on them and for example, you might want to set a smaller block size than the default because... If you know something of the structure of the file, and you know that there are records in the file of approximately n-bytes in length, approximately, then setting the block size a little bit smaller than n is a good idea. *[66m, 41s]*

"I guess my question was more related to the amount of RAM you require to do the file send. Does it always require 1% of the file size?"

No, it rolls along the file with a windowing thing; it doesn't require... it does require storing the signatures, but because it scales linearly, it has an upper bound on the memory it will use in that case. The main problem with the memory use of rsync is holding the file list. Because of the way I designed it... I didn't really design rsync for being used in the case of hundreds of millions of files with four megabyte machines, but people are using it that way. *[67m, 14s]*

And the problem is that the internal storage... I store the whole file list and sort it, because it's really convenient, having a sorted file list for a number of reasons, particularly hard links and things. I sort the file list, so I have to store it in memory and because of that... and the internal format uses approximately 100 bytes per file, so if you have a million files you are transferring, then it requires 100 megabytes of RAM, approximately, internally. Now that's expensive. On the wire, it only costs about 10 bytes per file, but I use an unpacked format for fast access in memory. *[67m, 52s]*

Now I'm planning on fixing that; I posted something about 18 months ago, a year ago or so, to the rsync mailing list of how I can fix this and it's a bit involved, but you can have a look on the rsync mailing list; there's a link to it on the rsync home page. There's a way around this, but unfortunately, it's fairly involved, which is partly why I've been avoiding doing it thus far; I've also been busy with various other things. But yeah, the memory usage is primarily in the holding of the file list; the size of the files is basically irrelevant to the memory usage of the program. Your page cache of course will be hit badly. *[68m, 24s]*

[The speaker calls on an audience member for a question.]

"I'm just curious with the renaming problem again; if you were to just send a file list at the beginning followed by a huge block of data corresponding to the contents of all the files... would that not just automatically solve the problem?"

Think about the order of magnitude of the problem of matching signatures to files. *[68m, 47s]*

“Well it would be equivalent of just sending a large tar file except you...”

No, it’s not. You also need a bigger block size to make it efficient. There are various things you can do, but none of them are particularly appealing. *[69m, 05s]*

[There is a comment from another audience member.]

“There’s a conflict here on the one hand of trying to send millions of files and do that efficiently in terms of overhead, and on the other hand, the desire some people have of sending a data distribution list of RPMs where there’s only a thousand files and handling renames.” *[69m, 27s]*

Yeah, you’re right. In the case of a thousand files, you could have an algorithm that could do much better with renames and yes, handling renames is something I do want to do, partly because of the stuff Steven has been doing with his apt-proxy stuff, where he’s actually done an rsync-based apt-proxy system. And look for it; I think it’s called apt-proxy, isn’t it Steven? Where are you? Yeah, he was not listening... [The audience laughs.] Have a look at freshmeat.net apt-proxy; handling renames and stuff like that would be cute. *[69m, 59s]*

The last few minutes, I’ll just get on with rzip. Rzip is the compression algorithm. This is primarily of academic interest. But it’s quite nice as far as the compression ratios it gets. Rzip is an rsync-inspired compression algorithm, but it’s not exactly the rsync algorithm inside it. But it has a lot in common with the rsync algorithm. Originally, it shared a lot of code, and then it didn’t for speed. *[70m, 20s]*

But it does give some very nice results. These are the compression ratios on various files... a particular version of emacs (you can get the versions out of my thesis), a particular version of Linux, my development home directory for Samba (so it’s got lots of different versions of Samba in the same (home) directory), and a mail archive. I collected spam from a 24-hour period on samba.org and notice, it’s 84 megabytes. [The audience laughs.] Unfortunately, samba@samba.org made it onto some spammer’s CD-ROM and so every spammer in the world hit us. And it was really quite awful. So this was the collected spam from a little period. *[70m, 57s]*

Anyway, it compresses very nicely. People told me that compressing spam was really easy; rm does it beautifully. [The audience laughs.] But anyway, if you do want to keep it for various analysis purposes, then this is great. So these are the compression ratios. Now notice that rzip wins in all cases. In some cases, rzip wins by a long, long way. *[71m, 16s]*

For example, my home directory, my development area, my Samba development area; gzip gets 3.50 compression ratio, bzip2 gets 4.78, rzip gets 8.93. Now if you know anything about the compression literature, you'll know that getting like a ten-percent improvement over something like bzip is considered good, getting a factor of two better is considered quite extraordinary. *[71m, 39s]*

The reason is that rzip, taking advantage of the rolling checksums and things, can use a massive history. gzip uses a 32K history; bzip, by default, uses a 900K history; rzip, by default, uses a 250 megabyte history. Right, so say you're backing up home directories, /home on a big system. And two people have got checked-out copies of the Linux kernel. And you rzip a tar file of /home. Then when it comes to the second copy of the Linux kernel tree, a hundred megabytes after the first one, it will say: oh, it's the same as this one back here with these small differences and a few bytes go out. *[72m, 22s]*

So it can find matching chunks of the order of hundreds of megabytes, going back a quarter gigabyte. And in fact you can make it more than a quarter gigabytes with some tweaks. And the way it does this... it's a bit like floating point; normal gzip-like algorithms... they're basically offset-length algorithms; they're based on: here's a match at offset this in the history and this length of match. Well what I do is the offset has a mantissa and exponent portion. Which means it is offset 2 to the power of this plus this. The rzip algorithm uses the rsync rolling checksums to find the matches. *[73m, 09s]*

And it's very, very, very slow. But if you need to compress really big files, and you think there's a lot of common data in it, like /home or spam file or something, then rzip is probably good for you. And it's available in the rzip CVS here. I've never released it to the world because it really is of academic interest, primarily. I've just told a few compression places about it, but it is available under the GPL if anyone wants to use it. *[73m, 34s]*

[The speaker calls on an audience member for a question.]

“How’s the decompression speed?”

Better than the compression, but still not good, but primarily because it’s lousy code, it’s academic code. [The audience laughs.] It’s written with instrumentation and things in it. As an example of what I mean by academic code in this particular case; it has built-in to it a bzip-like, because it uses bzip as its entropy encoder on the back end. My implementation of bzip is 50-60 times slower than the one in bzip2. It’s that sort of code. It’s code that’s clear but not particularly fast. So if anyone wants to speed it up, they can, but speeding up the compression to be actually acceptably fast, I think would be a hard task. *[74m, 17s]*

For more info on rsync, you can get it at rsync.samba.org; rsync.org isn’t my site. I have no idea what it points at... apparently there is an rsync.org, but anyway, somebody’s grabbed it. *[74m, 32s]*

linuxcare.com.au/rproxy if you want to hear about the rproxy accelerator

My home directory has got a copy of my thesis and things. There’s a tech report about rsync as well. Don’t read the tech report. Read the thesis instead. The thesis is clearer and more accurate. The tech report, we wrote when Paul and I were only just started working on this stuff. And Paul Mackerras is my supervisor at ANU; we worked on this stuff a bit together. So the thesis is a far more readable and accurate description of rsync. *[75m, 01s]*

And if you want to check out some of the code that isn’t released yet, like rzip and the latest version of rsync, then go to the CVS area on samba.org. *[75m, 11s]*

[The speaker calls on an audience member for a question.]

“A while ago, I think, you had some musings about problems of synchronizing many millions of files when there were not too many changes. I guess you had some plans for some kind of a hierarchical...”

Yes exactly, that’s what I described earlier when I said of the ways of reducing the memory and improving things for certain common cases. People were starting to do things like rsync export mirror.arnet.edu.au, this massive FTP site is equivalent of metalab in Australia, and exporting the whole tree and the load on the server was quite high when you start an rsync of the whole tree. And additionally, the memory requirements are far too high. *[76m, 01s]*

And it can be fixed. I just haven't fixed it yet. Hopefully, I'll fix it soon. I'm going to be using a little tdb database to store the cached copies of the directory trees, so I don't have to stat() all the inodes all the time and bring those pages into memory. And that will speed things up a lot as far as the launch time for scanning the file list and then I've got this hierarchical idea which unfortunately does add one round trip per element in the hierarchy but that will improve things a lot as far as the size of the file list in memory. Yeah, but it's not implemented yet, so I can't promise when it's going to be done. It just depends on what other things I'm working on. If somebody else wants to do it first, then submit the patch. *[76m, 44s]*

[The speaker calls on an audience member for a question.]

"Can't the server have precomputed signatures?"

No, because the client generates the signatures.

"What if you did it the other way around?"

You can do server-generated signatures and you could do precomputed server generated signatures. And there are some other reasons for doing server-generated signatures. The main ones being a patent concern. But unfortunately it requires basically a complete rewrite of rsync and it won't be backwards compatible. Paul and I had been considering doing that. We may do it at some stage and when we do go to server-generated signatures, we can start; we do have the possibility of doing caching of those. But while we're still doing client-generated signatures within the current infrastructure of rsync, you can't do it. It would basically be a new program. *[77m, 30s]*

[The speaker calls on an audience member for a question.]

"You told me you may be thinking of using a database of signatures to do an incremental backup using rsync..."

Yeah, I talk about that in my thesis a little bit. That's another application for rsync. You can do very nice incremental backup... it's ideal for like tape robots, tape robot backup. Say for example, you've got a silo, so you've got very, very slow access to the main data in your tape silo. But typically, you've got a fast front-end disk, which is, maybe, a tenth of a percent of the total size of your tape archive. *[78m, 10s]*

What you can do is: your initial backup you store to the tape, but then you store the signatures of the files on tape on the front-end disks. Then, when you do a subsequent backup, you can get just the differences without actually having to retrieve the old files off of tape. You just retrieve the signatures, which are small, off of disk, do the differencing using the rsync algorithm, then store the differences either to tape or to disk. Probably to tape because it's only a write; it'll be much faster than reading in all the data and it'll be small. And that allows you to do very nice incremental tape backups with very little tape activity. *[78m, 47s]*

And yeah, that's ideal. It would also be good for WORM drives, CD drives, where basically you store the original data and you just add a little bit of data on to the end of the WORM drive for the differences each time. It allows you to roll back to any version nicely while storing a minimum amount of data. You could make a really lovely backup system that way. So I'll expect the patches shortly. [The audience laughs.] And really it would be very nice. There was a company that was looking at doing it, I think, but it would be nice if there was a nice GPL'ed backup program that did this.

Okay, thanks very much.

[The audience applauds.] [The presentation ends.] *[79m, 39s]*

3. Additional resources

3.1. rsync

Details regarding the rsync application may be found at http://rsync.samba.org

A copy of Andrew Tridgell's Ph.D. thesis is available from http://samba.org/~tridge/

3.2. rproxy

Details regarding the rproxy project, protocols and application may be found at http://linuxcare.com.au/projects/rproxy/

3.3. Xdelta

Details regarding the Xdelta project, protocols and application may be found at http://www.XCF.Berkeley.EDU/~jmacd/xdelta.html

3.4. OpenSSH

Details regarding OpenSSH may be found at http://www.openssh.com

3.5. rzip

rzip and other unreleased code are available from the samba.org CVS; details at http://samba.org/cvs.html

3.6. mail scripts

The mail scripts mentioned in this talk (which use rsync) may be found at http://samba.org/pub/tridge/misc/mailscripts