远古超传
ULTRA DATA TRANSMISSION

Rsync增量传输场景

软 件 定 义 网 络　超 传 连 接 未 来

- Rsync 增量传输的性能可被近似建模为：

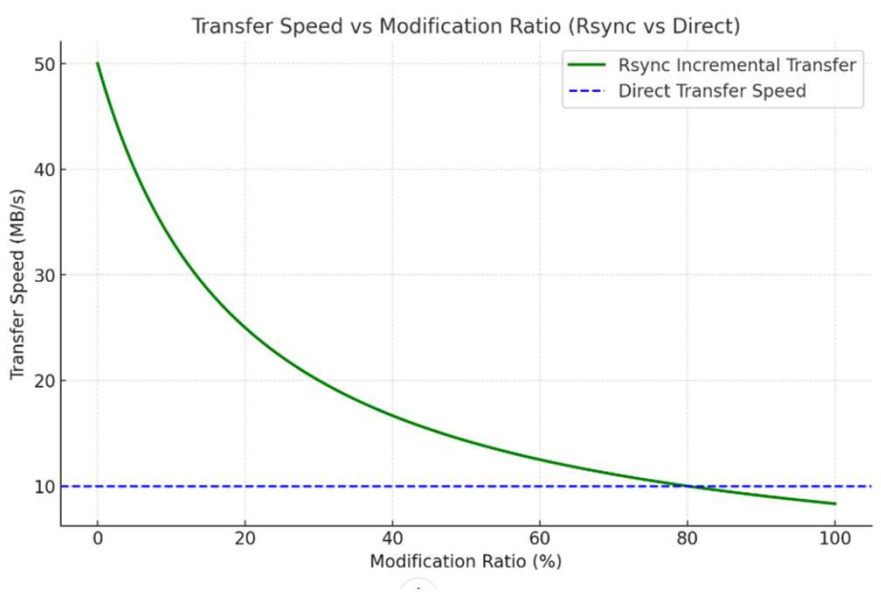
$$S_{\text{rsync}}(m) = S_{\text{base}} \cdot (1 - m) + S_{\text{delta}} \cdot m$$

其中：

  - S_{base} 是无修改块的传输速率。
  - S_{delta} 是修改块的传输速率。
  - m 是修改比例，范围 0-1。
- 蓝线的直接传输模型通常是一个常量 S_{direct}。

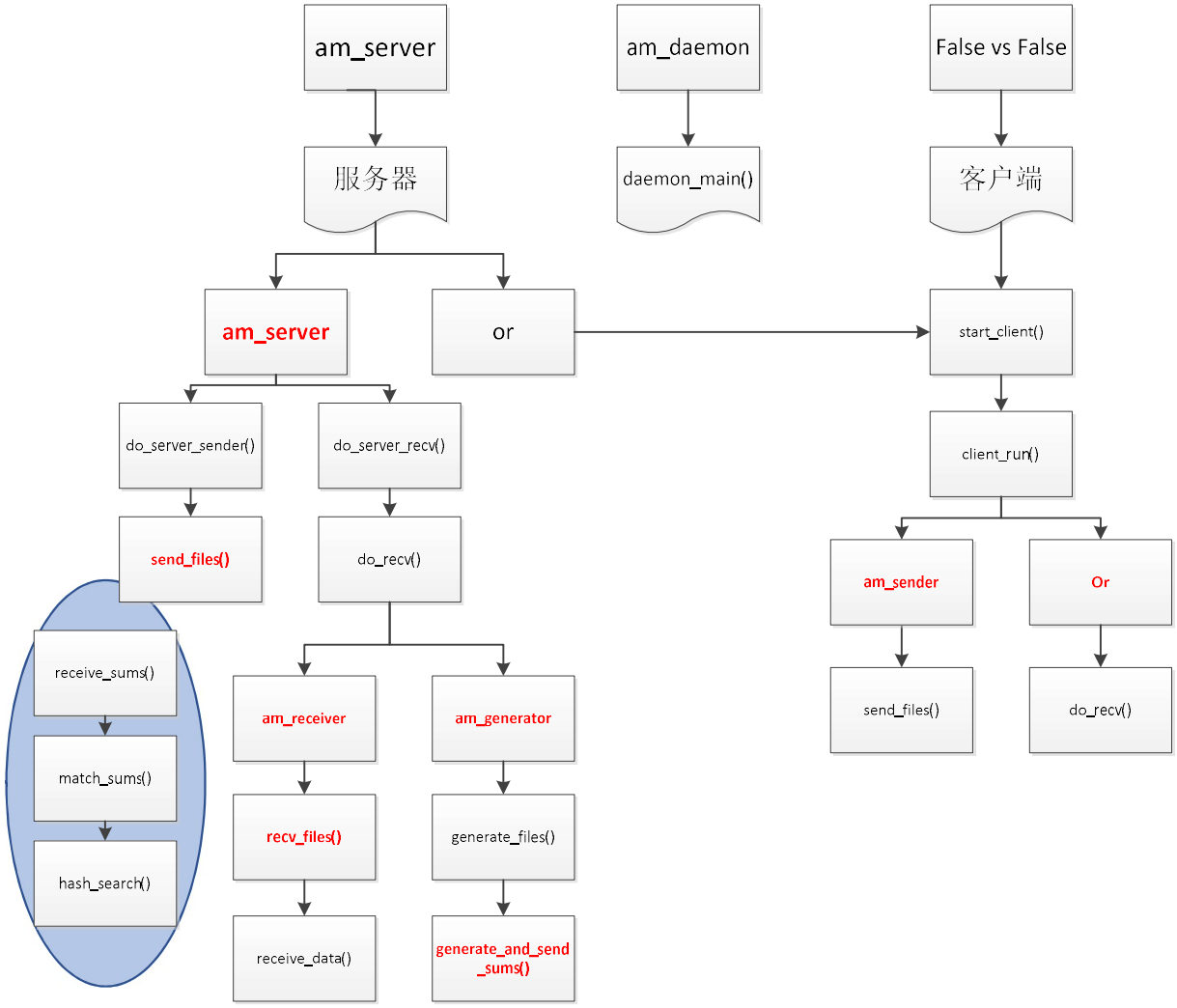
在该模型中：

$$S_{\text{base}} \approx 50\,\text{MB/s}, \quad S_{\text{delta}} \approx 10\,\text{MB/s}.$$

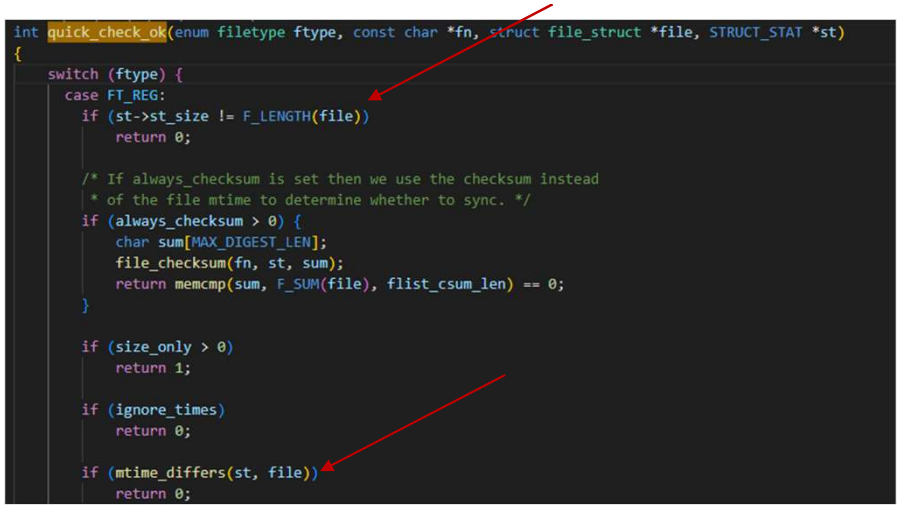


核心问题

- delta (encoding)change algorithm
- rolling checksum algorithm
- **The rsync algorithm**
  - 1. quick check (size, always checksum, mtime)
  - 2. rolling checksum in send_files()
  - 3. 3-level search in send_files()
- Server, Client and Daemon → sender & receiver & generator
  - send_files()/recv_files()/generate_files()


am_server
am_daemon
False vs False
服务器
daemon_main()
客户端
am_server
or
start_client()
do_server_sender()
do_server_recv()
client_run()
send_files()
do_recv()
am_sender
Or
receive_sums()
match_sums()
hash_search()
am_receiver
am_generator
send_files()
do_recv()
recv_files()
generate_files()
receive_data()
generate_and_send_sums()

1. quick_check_ok()

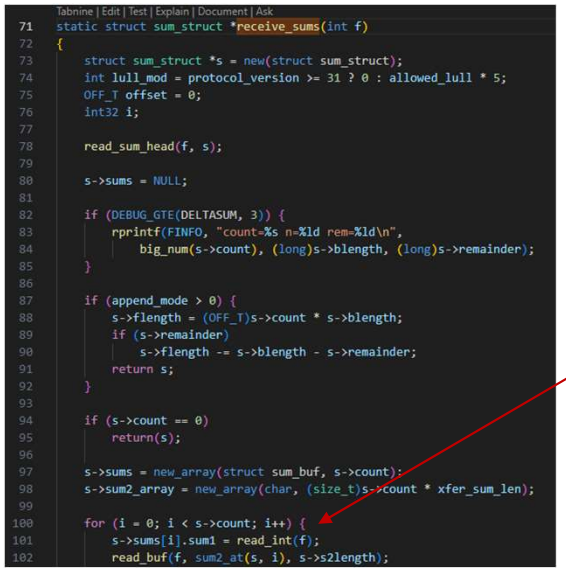


```c
int quick_check_ok(enum filetype ftype, const char *fn, struct file_struct *file, STRUCT_STAT *st)
{
    switch (ftype) {
      case FT_REG:
        if (st->st_size != F_LENGTH(file))
            return 0;

        /* If always_checksum is set then we use the checksum instead
         * of the file mtime to determine whether to sync. */
        if (always_checksum > 0) {
            char sum[MAX_DIGEST_LEN];
            file_checksum(fn, st, sum);
            return memcmp(sum, F_SUM(file), flist_csum_len) == 0;
        }

        if (size_only > 0)
            return 1;

        if (ignore_times)
            return 0;

        if (mtime_differs(st, file))
            return 0;
```

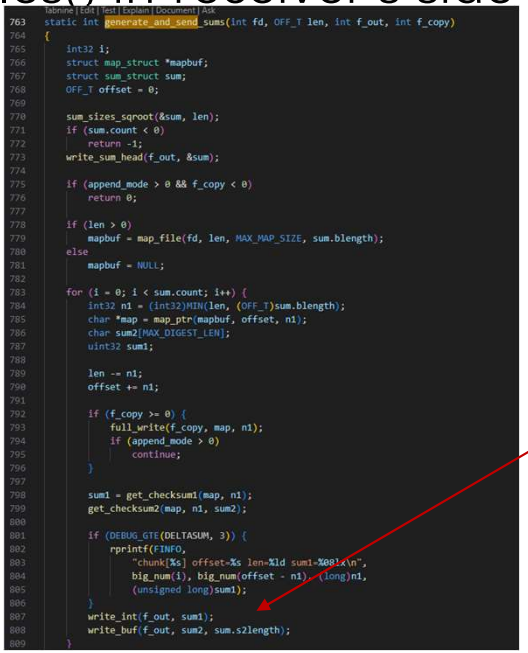
- **File Size:** If the sizes of the source and destination files differ, the file is marked for synchronization.
- **Always checksum:** Optionally, a full checksum of the file can be calculated for validation.
- **Modification time (mtime):** If the last modification times of the files differ, rsync flags the file for further checks or synchronization.

sum functions

- receive_sums(): called in send_files() to receive checksums from receiver
- match_sums(): called in send_files() to match checksums from receiver
- generate_and_send_sums(): called in generate_files() in receiver's side
- sum_update()
- get_checksum1()
- get_checksum2()
- sum_end()
- sum_init()
- …



```
Tabnine | Edit | Test | Explain | Document | Ask
71  static struct sum_struct *receive_sums(int f)
72  {
73      struct sum_struct *s = new(struct sum_struct);
74      int lull_mod = protocol_version >= 31 ? 0 : allowed_lull * 5;
75      OFF_T offset = 0;
76      int32 i;
77
78      read_sum_head(f, s);
79
80      s->sums = NULL;
81
82      if (DEBUG_GTE(DELTASUM, 3)) {
83          rprintf(FINFO, "count=%s n=%ld rem=%ld\n",
84              big_num(s->count), (long)s->blength, (long)s->remainder);
85      }
86
87      if (append_mode > 0) {
88          s->flength = (OFF_T)s->count * s->blength;
89          if (s->remainder)
90              s->flength -= s->blength - s->remainder;
91          return s;
92      }
93
94      if (s->count == 0)
95          return(s);
96
97      s->sums = new_array(struct sum_buf, s->count);
98      s->sum2_array = new_array(char, (size_t)s->count * xfer_sum_len);
99
100     for (i = 0; i < s->count; i++) {
101         s->sums[i].sum1 = read_int(f);
102         read_buf(f, sum2_at(s, i), s->s2length);
```



```
Tabnine | Edit | Test | Explain | Document | Ask
763 static int generate_and_send_sums(int fd, OFF_T len, int f_out, int f_copy)
764 {
765     int32 i;
766     struct map_struct *mapbuf;
767     struct sum_struct sum;
768     OFF_T offset = 0;
769
770     sum_sizes_sqroot(&sum, len);
771     if (sum.count < 0)
772         return -1;
773     write_sum_head(f_out, &sum);
774
775     if (append_mode > 0 && f_copy < 0)
776         return 0;
777
778     if (len > 0)
779         mapbuf = map_file(fd, len, MAX_MAP_SIZE, sum.blength);
780     else
781         mapbuf = NULL;
782
783     for (i = 0; i < sum.count; i++) {
784         int32 n1 = (int32)MIN(len, (OFF_T)sum.blength);
785         char *map = map_ptr(mapbuf, offset, n1);
786         char sum2[MAX_DIGEST_LEN];
787         uint32 sum1;
788
789         len -= n1;
790         offset += n1;
791
792         if (f_copy >= 0) {
793             full_write(f_copy, map, n1);
794             if (append_mode > 0)
795                 continue;
796         }
797
798         sum1 = get_checksum1(map, n1);
799         get_checksum2(map, n1, sum2);
800
801         if (DEBUG_GTE(DELTASUM, 3)) {
802             rprintf(FINFO,
803                 "chunk[%s] offset=%s len=%ld sum1=%08lx\n",
804                 big_num(i), big_num(offset - n1), (long)n1,
805                 (unsigned long)sum1);
806         }
807         write_int(f_out, sum1);
808         write_buf(f_out, sum2, sum.s2length);
809     }
```

2. match_sums()

```
391     if (len > 0 && s->count > 0) {
392         build_hash_table(s);
393
394         if (DEBUG_GTE(DELTASUM, 2))
395             rprintf(FINFO,"built hash table\n");
396
397         hash_search(f, s, buf, len);
398
399         if (DEBUG_GTE(DELTASUM, 2))
400             rprintf(FINFO,"done hash search\n");
401     } else {
402         OFF_T j;
403         /* by doing this in pieces we avoid too many seeks */
404         for (j = last_match + CHUNK_SIZE; j < len; j += CHUNK_SIZE)
405             matched(f, s, buf, j, -2);
406         matched(f, s, buf, len, -1);
407     }
```

- Files are divided into fixed-size blocks
- A **rolling checksum** is calculated for each block in the source file. This checksum allows quick recalculations when the comparison window moves by one byte, **using a formula** that updates the checksum incrementally rather than recomputing it from scratch.
- The **destination file's blocks** have both a rolling (weak) checksum and a strong checksum (e.g., MD5) stored. The weak checksum quickly identifies potential matches, and the strong checksum **confirms** these matches.

get_checksum1()/get_checksum2()

The weak checksum algorithm we used in our implementation was inspired by Mark Adler's adler-32 checksum. Our checksum is defined by

```
/*
  a simple 32 bit checksum that can be updated from either end
  (inspired by Mark Adler's Adler-32 checksum)
  */
Tabnine | Edit | Test | Explain | Document | Ask
uint32 get_checksum1(char *buf1, int32 len)
{
    int32 i;
    uint32 s1, s2;
    schar *buf = (schar *)buf1;

    s1 = s2 = 0;
    for (i = 0; i < (len-4); i+=4) {
        s2 += 4*(s1 + buf[i]) + 3*buf[i+1] + 2*buf[i+2] + buf[i+3] + 10*CHAR_OFFSET;
        s1 += (buf[i+0] + buf[i+1] + buf[i+2] + buf[i+3] + 4*CHAR_OFFSET);
    }
    for (; i < len; i++) {
        s1 += (buf[i]+CHAR_OFFSET); s2 += s1;
    }
    return (s1 & 0xffff) + (s2 << 16);
}
```

$$a(k,l) = (\sum_{i=k}^{l} X_i) \bmod M$$

$$b(k,l) = (\sum_{i=k}^{l} (l-i+1)X_i) \bmod M$$

$$s(k,l) = a(k,l) + 2^{16}\, b(k,l)$$

where $s(k,l)$ is the rolling checksum of the bytes $X_k \ldots X_l$. For simplicity and speed, we use $M = 2^{16}$.

Hash check not hitted

Incremental Rolling Checksum Formula

For a block of size k:

1. **Checksum Initialization:**

$$s1 = \sum_{i=1}^{k} \text{block}[i]$$

$$s2 = \sum_{i=1}^{k} (k - i + 1) \cdot \text{block}[i]$$

2. **Rolling Update:** When the window shifts one byte:
   - Remove the first byte (`map[0]`) and add the new byte (`map[k]` if available):

$$s1_{\text{new}} = s1_{\text{old}} - \text{block}[0] + \text{block}[k]$$

$$s2_{\text{new}} = s2_{\text{old}} - k \cdot \text{block}[0] + s1_{\text{new}}$$

```
null_hash:
  backup = (int32)(offset - last_match);
  /* We sometimes read 1 byte prior to last_match... */
  if (backup < 0)
      backup = 0;

  /* Trim off the first byte from the checksum */
  more = offset + k < len;
  map = (schar *)map_ptr(buf, offset - backup, k + more + backup) + backup;
  s1 -= map[0] + CHAR_OFFSET;
  s2 -= k * (map[0]+CHAR_OFFSET);

  /* Add on the next byte (if there is one) to the checksum */
  if (more) {
      s1 += map[k] + CHAR_OFFSET;
      s2 += s1;
  } else
      --k;

  /* By matching early we avoid re-reading the
     data 3 times in the case where a token
     match comes a long way after last
     match. The 3 reads are caused by the
     running match, the checksum update and the
     literal send. */
  if (backup >= s->blength+CHUNK_SIZE && end-offset > CHUNK_SIZE)
      matched(f, s, buf, offset - s->blength, -2);
} while (++offset < end);
```

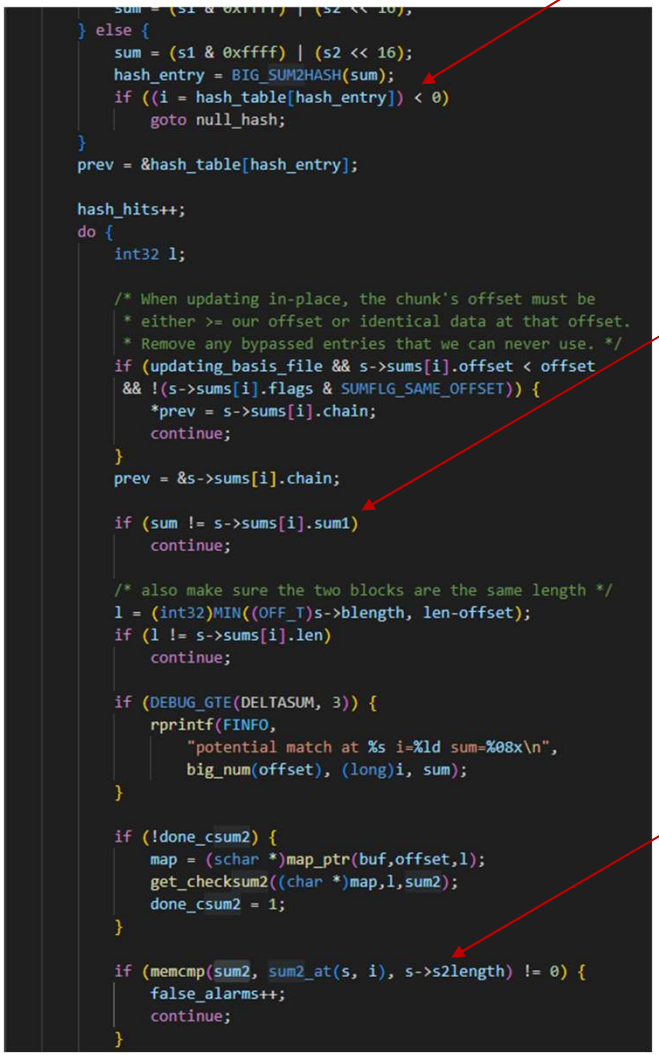
$$a(k+1,l+1) = (a(k,l) - X_k + X_{l+1}) \bmod M$$

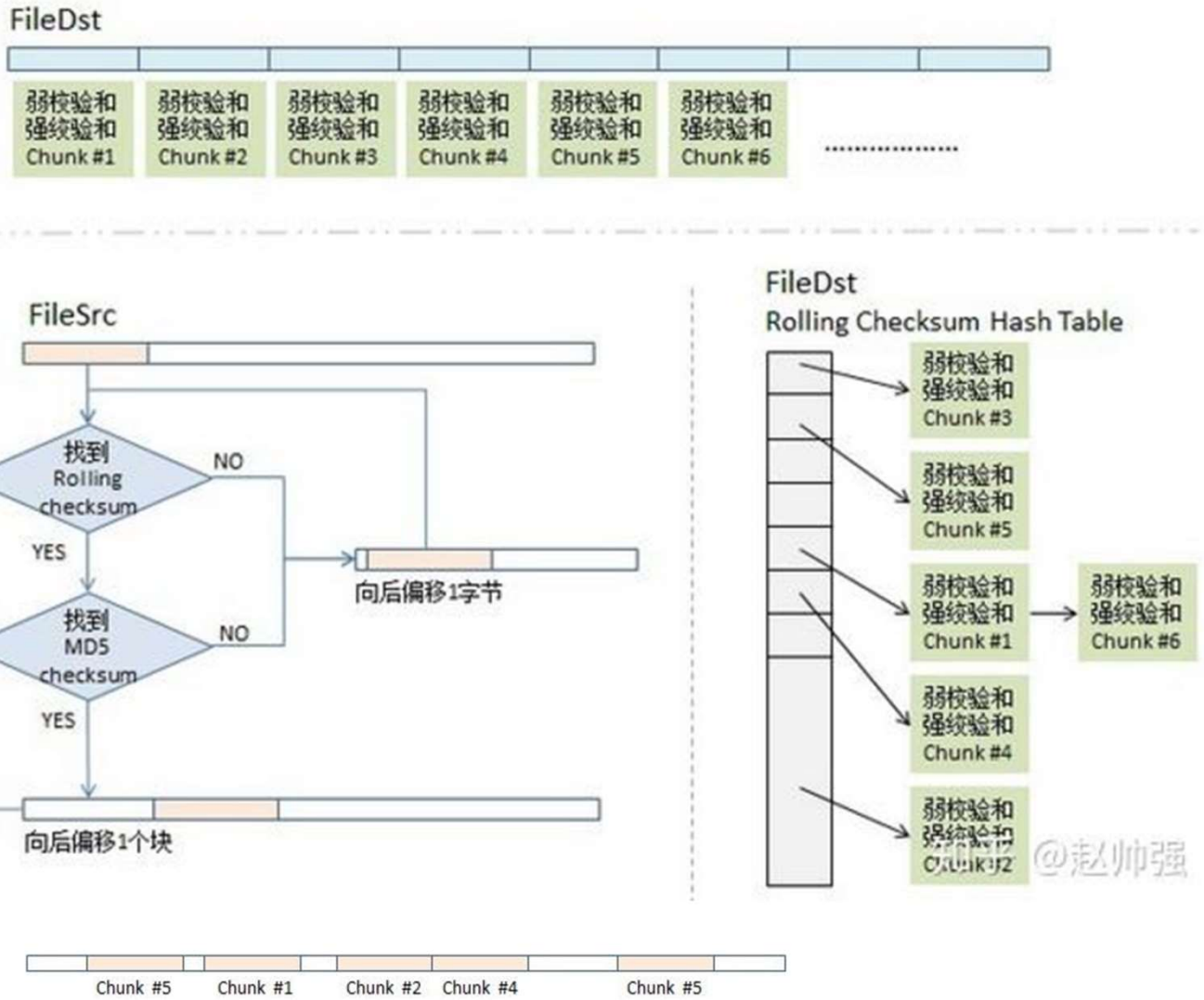
$$b(k+1,l+1) = (b(k,l) - (l-k+1)X_k + a(k+1,l+1)) \bmod M$$

Thus the checksum can be calculated for blocks of length S at all possible offsets within a file in a ``rolling" fashion, with very little computation at each point.

3. hash_search()

- Level 1: 16-bit hash table:
  - A hash of the rolling checksum is calculated and used to index a hash table of block checksums.
  - The hash table reduces the number of blocks to consider.
- Level 2: Weak checksum comparison:
  - If the hash table indicates a match, the weak checksum of the current source block is compared to the destination's checksum.
- Level 3: Strong checksum validation:
  - For potential matches, the strong checksum (e.g., MD5) confirms the block match with near certainty.

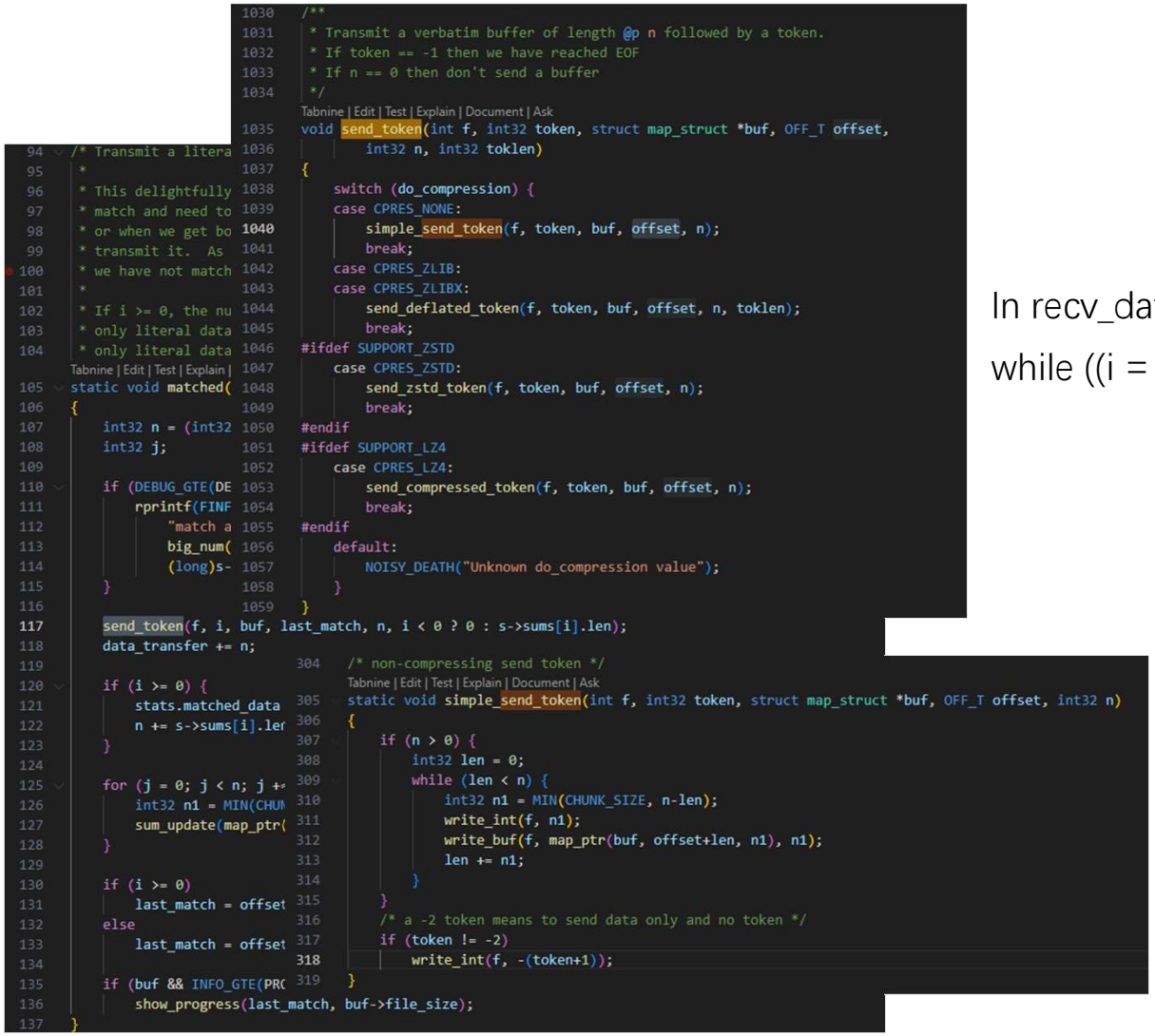

} else {
sum = (s1 & 0xffff) | (s2 << 16);
hash_entry = BIG_SUM2HASH(sum);
if ((i = hash_table[hash_entry]) < 0)
goto null_hash;
}
prev = &hash_table[hash_entry];
hash_hits++;
do {
int32 l;
/* When updating in-place, the chunk's offset must be
* either >= our offset or identical data at that offset.
* Remove any bypassed entries that we can never use. */
if (updating_basis_file && s->sums[i].offset < offset
&& !(s->sums[i].flags & SUMFLG_SAME_OFFSET)) {
*prev = s->sums[i].chain;
continue;
}
prev = &s->sums[i].chain;
if (sum != s->sums[i].sum1)
continue;
/* also make sure the two blocks are the same length */
l = (int32)MIN((OFF_T)s->blength, len-offset);
if (l != s->sums[i].len)
continue;
if (DEBUG_GTE(DELTASUM, 3)) {
rprintf(FINFO,
"potential match at %s i=%ld sum=%08x\n",
big_num(offset), (long)i, sum);
}
if (!done_csum2) {
map = (schar *)map_ptr(buf,offset,l);
get_checksum2((char *)map,l,sum2);
done_csum2 = 1;
}
if (memcmp(sum2, sum2_at(s, i), s->s2length) != 0) {
false_alarms++;
continue;
}


FileDst
弱校验和
强校验和
Chunk #1
弱校验和
强校验和
Chunk #2
弱校验和
强校验和
Chunk #3
弱校验和
强校验和
Chunk #4
弱校验和
强校验和
Chunk #5
弱校验和
强校验和
Chunk #6
FileSrc
找到
Rolling
checksum
NO
YES
找到
MD5
checksum
NO
YES
向后偏移1字节
向后偏移1个块
FileDst
Rolling Checksum Hash Table
弱校验和
强校验和
Chunk #3
弱校验和
强校验和
Chunk #5
弱校验和
强校验和
Chunk #1
弱校验和
强校验和
Chunk #6
弱校验和
强校验和
Chunk #4
弱校验和
强校验和
Chunk #2
@赵帅强
Chunk #5
Chunk #1
Chunk #2
Chunk #4
Chunk #5

Multiple hits in hash table

```
240         /* When updating in-place, the best possible match is
241          * one with an identical offset, so we prefer that over
242          * the adjacent want_i optimization. */
243         if (updating_basis_file) {
244             /* All the generator's chunks start at blength boundaries. */
245             while (aligned_offset < offset) {
246                 aligned_offset += s->blength;
247                 aligned_i++;
248             }
249             if ((offset == aligned_offset
250               || (sum == 0 && l == s->blength && aligned_offset + l <= len))
251              && aligned_i < s->count) {
252                 if (i != aligned_i) {
253                     if (sum != s->sums[aligned_i].sum1
254                      || l != s->sums[aligned_i].len
255                      || memcmp(sum2, sum2_at(s, aligned_i), s->s2length) != 0)
256                         goto check_want_i;
257                     i = aligned_i;
258                 }
259                 if (offset != aligned_offset) {
260                     /* We've matched some zeros in a spot that is also zeros
261                      * further along in the basis file, if we find zeros ahead
262                      * in the sender's file, we'll output enough literal data
263                      * to re-align with the basis file, and get back to seeking
264                      * instead of writing. */
265                     backup = (int32)(aligned_offset - last_match);
266                     if (backup < 0)
267                         backup = 0;
268                     map = (schar *)map_ptr(buf, aligned_offset - backup, l + backup)
269                         + backup;
270                     sum = get_checksum1((char *)map, l);
271                     if (sum != s->sums[i].sum1)
272                         goto check_want_i;
273                     get_checksum2((char *)map, l, sum2);
274                     if (memcmp(sum2, sum2_at(s, i), s->s2length) != 0)
275                         goto check_want_i;
276                     /* OK, we have a re-alignment match.  Bump the offset
277                      * forward to the new match point. */
278                     offset = aligned_offset;
279                 }
280                 /* This identical chunk is in the same spot in the old and new file. */
281                 s->sums[i].flags |= SUMFLG_SAME_OFFSET;
282                 want_i = i;
283             }
284         }
```

```
286         check_want_i:
287           /* we've found a match, but now check to see
288            * if want_i can hint at a better match. */
289           if (i != want_i && want_i < s->count
290            && (!updating_basis_file || s->sums[want_i].offset >= offset
291             || s->sums[want_i].flags & SUMFLG_SAME_OFFSET)
292            && sum == s->sums[want_i].sum1
293            && memcmp(sum2, sum2_at(s, want_i), s->s2length) == 0) {
294               /* we've found an adjacent match - the RLL coder
295                * will be happy */
296               i = want_i;
297           }
298           want_i = i + 1;
299
300           matched(f,s,buf,offset,i);
301           offset += s->sums[i].len - 1;
302           k = (int32)MIN((OFF_T)s->blength, len-offset);
303           map = (schar *)map_ptr(buf, offset, k);
304           sum = get_checksum1((char *)map, k);
305           s1 = sum & 0xFFFF;
306           s2 = sum >> 16;
307           matches++;
308           break;
309         } while ((i = s->sums[i].chain) >= 0);
```



```
/* Transmit a litera
 *
 * This delightfully
 * match and need to
 * or when we get bo
 * transmit.  As
 * we have not match
 *
 * If i >= 0, the nu
 * only literal data
 * only literal data
Tabnine | Edit | Test | Explain |
static void matched(
{
    int32 n = (int32
    int32 j;

    if (DEBUG_GTE(DE
        rprintf(FINF
            "match a
            big_num(
            (long)s-
    }

    send_token(f, i, buf, last_match, n, i < 0 ? 0 : s->sums[i].len);
    data_transfer += n;

    if (i >= 0) {
        stats.matched_data
        n += s->sums[i].le
    }

    for (j = 0; j < n; j +
        int32 n1 = MIN(CHU
        sum_update(map_ptr(
    }

    if (i >= 0)
        last_match = offset
    else
        last_match = offset

    if (buf && INFO_GTE(PR
        show_progress(last_match, buf->file_size);
}
```

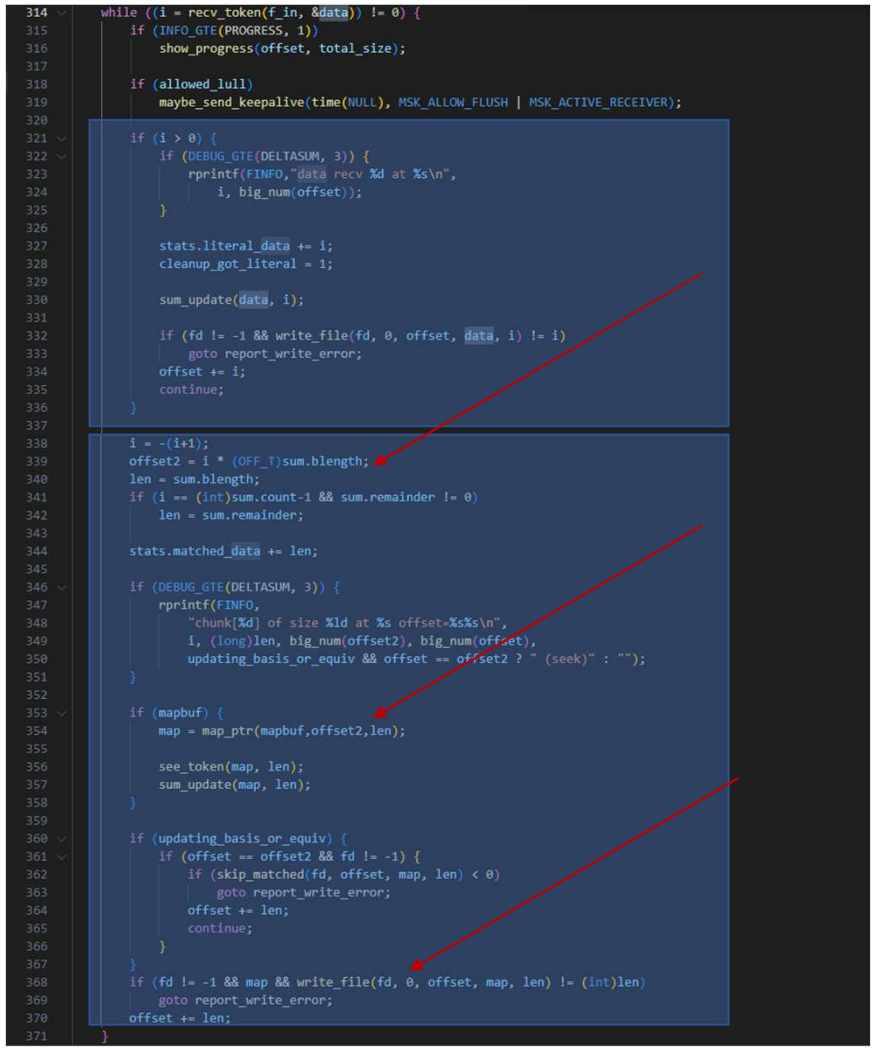
```
/**
 * Transmit a verbatim buffer of length @p n followed by a token.
 * If token == -1 then we have reached EOF
 * If n == 0 then don't send a buffer
 */
Tabnine | Edit | Test | Explain | Document | Ask
void send_token(int f, int32 token, struct map_struct *buf, OFF_T offset,
                int32 n, int32 toklen)
{
    switch (do_compression) {
    case CPRES_NONE:
        simple_send_token(f, token, buf, offset, n);
        break;
    case CPRES_ZLIB:
    case CPRES_ZLIBX:
        send_deflated_token(f, token, buf, offset, n, toklen);
        break;
#ifdef SUPPORT_ZSTD
    case CPRES_ZSTD:
        send_zstd_token(f, token, buf, offset, n);
        break;
#endif
#ifdef SUPPORT_LZ4
    case CPRES_LZ4:
        send_compressed_token(f, token, buf, offset, n);
        break;
#endif
    default:
        NOISY_DEATH("Unknown do_compression value");
    }
}
```

```
/* non-compressing send token */
Tabnine | Edit | Test | Explain | Document | Ask
static void simple_send_token(int f, int32 token, struct map_struct *buf, OFF_T offset, int32 n)
{
    if (n > 0) {
        int32 len = 0;
        while (len < n) {
            int32 n1 = MIN(CHUNK_SIZE, n-len);
            write_int(f, n1);
            write_buf(f, map_ptr(buf, offset+len, n1), n1);
            len += n1;
        }
    }
    /* a -2 token means to send data only and no token */
    if (token != -2)
        write_int(f, -(token+1));
}
```

In recv_data():

while ((i = recv_token(f_in, &data)) != 0){…}

```
/* non-compressing recv token */
Tabnine | Edit | Test | Explain | Document | Ask
static int32 simple_recv_token(int f, char **data)
{
    static int32 residue;
    static char *buf;
    int32 n;

    if (!buf)
        buf = new_array(char, CHUNK_SIZE);

    if (residue == 0) {
        int32 i = read_int(f);
        if (i <= 0)
            return i;
        residue = i;
    }

    *data = buf;
    n = MIN(CHUNK_SIZE,residue);
    residue -= n;
    read_buf(f,buf,n);
    return n;
}
```



```
while ((i = recv_token(f_in, &data)) != 0) {
    if (INFO_GTE(PROGRESS, 1))
        show_progress(offset, total_size);

    if (allowed_lull)
        maybe_send_keepalive(time(NULL), MSK_ALLOW_FLUSH | MSK_ACTIVE_RECEIVER);

    if (i > 0) {
        if (DEBUG_GTE(DELTASUM, 3)) {
            rprintf(FINFO,"data recv %d at %s\n",
                i, big_num(offset));
        }

        stats.literal_data += i;
        cleanup_got_literal = 1;

        sum_update(data, i);

        if (fd != -1 && write_file(fd, 0, offset, data, i) != i)
            goto report_write_error;
        offset += i;
        continue;
    }

    i = -(i+1);
    offset2 = i * (OFF_T)sum.blength;
    len = sum.blength;
    if (i == (int)sum.count-1 && sum.remainder != 0)
        len = sum.remainder;

    stats.matched_data += len;

    if (DEBUG_GTE(DELTASUM, 3)) {
        rprintf(FINFO,
            "chunk[%d] of size %ld at %s offset=%s%s\n",
            i, (long)len, big_num(offset2), big_num(offset),
            updating_basis_or_equiv && offset == offset2 ? " (seek)" : "");
    }

    if (mapbuf) {
        map = map_ptr(mapbuf,offset2,len);

        see_token(map, len);
        sum_update(map, len);
    }

    if (updating_basis_or_equiv) {
        if (offset == offset2 && fd != -1) {
            if (skip_matched(fd, offset, map, len) < 0)
                goto report_write_error;
            offset += len;
            continue;
        }
    }
    if (fd != -1 && map && write_file(fd, 0, offset, map, len) != (int)len)
        goto report_write_error;
    offset += len;
}
```

核心问题

- delta (encoding)change algorithm
- rolling checksum algorithm
- **The rsync algorithm**
  - 1. quick check (size, always checksum, mtime)
  - 2. rolling checksum in send_files()
  - 3. 3-level search in send_files()
- Server, Client and Daemon → sender & receiver & generator
  - send_files()/recv_files()/generate_files()

谢 谢!

THANKS

远古超传
ULTRADATA TRANSMISSION

软 件 定 义 网 络　超 传 连 接 未 来