Managing self-inflicted nondeterminism

Dmitrii Zagorodnov
Dept of Computer Science
University of Tromsø
dmitrii@cs.uit.no

Keith Marzullo
Dept of Computer Science & Engineering
University of California, San Diego
marzullo@cs.ucsd.edu

1. Introduction

When a computation is replicated for greater availability – either by simultaneous execution on multiple machines or by restarting it after a crash – the problem of keeping replicas consistent arises. Replicas lose consistency when their run-time states diverge. This can happen because of differences in hardware inputs (clock ticks, network packets, keystrokes, etc.), which in turn lead to differences in "wall clock" time and in scheduling of events, such as context switches and signals. Hence, asynchrony in hardware makes software nondeterministic.

When replicating existing services, one does not have the ability to change the service to be flexible with respect to nondeterminism. Given this constraint, the traditional approach to maintaining replica consistency is to impose tighter synchronization. By forcing replicas to operate in lockstep – either on the level of hardware signals [8] or on the level of processor instructions [4] – the entire memory state of replicated machines can be kept identical. Working on a higher level, some systems [2, 6, 3] synchronize only the state that is deemed important for consistent execution. Since it is difficult to precisely identify the relevant sources of nondeterminism, these systems are conservative by synchronizing much more than is necessary.

The cost of tighter synchronization is a loss of performance. For example, Hypervisor [4] suffered roughly a factor of 2 overhead in execution and TFT [3] reported between 23% and 58% for gzip, depending on the compression level. Although the cost of synchronization decreases when it is performed at higher levels, the cost of engineering a high-level solution grows since changing the OS or the application is usually time-consuming and error-prone.

Our experience in replicating TCP-based network servers [1, 9] indicates that the overwhelming majority of state inconsistencies among replicas never lead to differences in the external behavior of the server. One can draw an analogy between the effects of a nondeterministic event (such as a hardware interrupt) and the effects of a *fault* (a hardware glitch or a software bug): both have the potential to cause the service to diverge from its specification, but neither is guaranteed to do so. Not every fault leads to an *error* in the state and, likewise, not every event leads to a state divergence. Even when there is an error or a divergence in the run-time state, it does not always "leak outside" and cause a *failure* of the service. When it does not, we call the event *benign*; otherwise, we say that it is *malignant*.

Furthermore, our experience with several applications indicates that the most prevalent malignant state inconsistencies in network servers are the result of explicit nondeterminism in the service protocol itself. For example, when a video player opens a stream, the server returns a handle for the stream. This handle is a random value; different replicas will generate different handles. We call such nondeterminism *self-inflicted*. By definition, self-inflicted nondeterminism is malignant.

We argue that synchronizing replicas that suffer from self-inflicted nondeterminism is best done with an application-specific interceptor that manipulates the I/O streams of the replicas so as to mask non-benign differences among their states. Instead of forcing replicas to have consistent states, we patch up the differences when they leak outside (e.g. by adjusting all occurrences of handles in the I/O stream). This approach combines the benefits of a low-level solution (no changes to the OS or the application are required) with the performance advantage of a high-level solution (it only resolves the states of replicas that are guaranteed to be malignant).

Our approach incurs the cost of implementing an interceptor for every application. This cost can be kept low, however, by relying on a general architecture for performing stream manipulations, which we present in the next section. We illustrate the use of this architecture with an example interceptor for a popular application-level protocol (SMB) in Section 3, which concludes with the description of the two sources of self-inflicted nondeterminism that we encountered in the Samba server. In Section 4 we evaluate the overhead of interception on communication throughput, and in Section 5 we conclude with a discussion of related and future work.

2. Architecture

Our approach uses interception of the I/O streams of replicas. The interceptors change divergent values for data leaving the server and remap the same values to their different, replica-specific values when coming into the server. In the following we consider only TCP/IP-based communication, but the method can be trivially altered for other transports.

2.1. Stream Parsing

A packet consists of data encapsulated within multiple levels of headers. At each level there is a *header* and the associated encapsulated data, which we call the *remainder*. The steps for "unwrapping" the packet are conceptually the same at each level: first, a fixed-length header is parsed; then, based on the information in the header, the remaining data is processed. This may include scanning for additional headers that were added after the main one (e.g. TCP options) and checking the integrity of the payload (e.g. verifying a checksum). The remainder is then passed to either the next higher or the lower level, depending on the direction of the stream. At each level the sizes of the basic header (without options) and the remaining data are known: the header size is prescribed by the protocol specification and the data size can be either learned from the header or from the level that gave it the data.

Since headers are necessarily well-formed, a protocol-specific filter can locate and change fields in the data stream. In an earlier paper [1], we showed how a "wrapper" around the TCP/IP stack can be used to reconcile differences in the states of network protocol drivers on replicas. By manipulating fields in the packet header (TCP sequence numbers, TCP windows, and IP addresses), we enabled a backup replica to communicate with the client after a crash of the primary, even though the backup's notions of the sequence number and the IP address are different. This technique can be generalized and applied to application-level protocols, too, as an efficient solution to self-induced nondeterminism.

Conceptually, our architecture allows one to insert an interceptor at any level in the protocol hierarchy in which nondeterministic values occur. These interceptors obtain input just before it gets to the receiver (which is a protocol or an application) and they obtain the output right after it is sent. In the case of application-level protocols, this mapping of values occurs at the system call boundary, where the data flows between the protocol stack and the application.

An interceptor may manipulate any field in the data segment, but it may *not* increase its size. An interceptor can maintain state for each session and any part of that state may be exchanged with interceptors on other replicas for coordination purposes.

2.2. Structuring Constraints

In this paper we focus on the application level. A wrapper technique can be used for other layers [1].

If a recv() system call always returned a complete packet, then modifying data with an interceptor would be straightforward. In reality, there are several structuring constraints:

- **Fragmentation:** Unlike the lower-level packet-based protocols, TCP provides a stream abstraction and as such it can return data in chunks of any size.
- **Chaining:** Certain application-level protocols may chain several requests together by sending them back to back, with all but the last one indicating that there is another request. For instance, Samba's SMB protocol chains using AndX-type requests, as will be described in Section 3.
- **Nesting:** Data that comes out of a TCP socket may contain several layers of headers all of which are processed inside the same server application. For example, SMB messages are encapsulated inside NBT protocol messages, both of which are processed by the Samba dæmon.

In our implementation, a generic protocol-agnostic interception routine invokes a protocol-specific routine to parse a header. Accommodating fragmentation, which amounts to waiting for enough data to collect to form a unit that can be parsed, is done in the generic interceptor, which can wait for more inbound data by blocking the recv(), or collect more outbound data from several subsequent send()s.

A difficulty is determining how much data the interceptor should wait for before passing it on. For example, in the case of inbound data, waiting for more than the client can send before it blocks would stall the connection. Also, even if the client never blocks, processing data in chunks that are too large may introduce additional latency.

We chose the following strategy: if we are expecting a header, then we wait for the full header to arrive. (If fewer bytes arrive on an incoming stream, we make additional recv() calls; if the application has not sent enough data, we wait for more send() calls.) If instead we are expecting a remainder, then we process it in whatever chunks it arrives, with the exception of special cases (described below), in which the protocol handler requests the remainder to be processed in multiples of several bytes. If the chunk that we chose to process is larger than the buffer supplied with recv(), the remaining data will be queued until the next invocation of recv(). Although this queuing can introduce additional latency, we do expect such situations to arise quite rarely, since applications also tend to read streams in chunks that are either equal to protocol header size or are large enough to receive a typical application-level packet. For the same reason, applications that are sensitive to the timing of packets should not be affected by our strategy.

To deal with chaining and nesting we allow multiple levels of protocol-specific routines, which we call *parsers*. If a top-level parser – the one that gets invoked when there is traffic on a specific port – determines that the remainder encapsulates a higher-level protocol message, then it can return a pointer to a different parser for processing that message. The generic interceptor then knows to invoke that other parser for all the data in the remainder.

Chaining is handled by this approach as follows: if the main protocol header is followed by several chained requests, each carrying a small additional header (as it happens in SMB), then the parser for the main header returns a pointer to another parser that is invoked on each "chained" message in the remainder. The next section makes all of this more concrete and Section 3 presents an example.

2.3. Writing a parser

A protocol-specific parser receives two parameters: a reference to the data and its size. The size parameter is unnecessary when the parser knows what kind of header it is dealing with, but it can be useful for a parser operating on unstructured data. The parser also has access to the connection-specific state which is maintained between invocations, not unlike private variables of a class instance.

A parser returns three values: the length of the remainder, the remainder specification, and the remainder parser function reference. The first value is either a non-negative integer or a special value *unknown*, indicating that the size of the remainder could not be determined from the header. In that case the general interceptor may be able to determine the remainder size using the value returned previously by the lower-level parser. For all protocols it should be possible to determine the remainder size in some manner, since the appropriate protocol layer needs to be able to wrap or unwrap a packet. The remainder spec may contain the following values:

- **Unrestricted:** This means the remainder may be processed in chunks of arbitrary size. Most likely this value will be accompanied by a NULL reference for the parser function, meaning that no processing on the remainder is needed, so it just gets passed to the next layer immediately.
- **Multiples of size M:** This means that the remainder should be processed in multiples of M bytes. This allows passing the data stream through a filter that decrypts, changes something, and re-encrypts the stream. Since encryption usually operates on blocks larger than a byte, it is easier to implement such parsers if data chunks are always aligned with the block boundaries.
- **Header of size N:** This means that the remainder encapsulates a higher-level protocol with the header of size N. This value should be accompanied by a reference to a parser for the protocol that will be responsible for handling all data in the remainder.

The general interceptor allows registration of protocol-specific parsers and calls upon them when needed. Registering a parser associates the parser routine reference and a standard header size to a port. Our implementation allows one to attach a different parser to each direction, but normally the same one is used for both. For each new connection the general interceptor checks whether the server port of that connection has a parser registered with it. If it does, then the general interceptor allocates an interceptor structure consisting of the reference to the parser, the standard header size, the amount of bytes in the remainder (initially zero), the state reference, and a reference to another, higher-level interceptor structure (initially NULL).

The last field allows the general interceptor to arrange interceptor structures in a linked list as follows: If a parser returns a reference to another parser, then

a new interceptor structure is linked to the current one and that parser is called for as long as there is data in the current remainder. When no bytes are left in the current remainder, the last interceptor structure is removed from the list and processing resumes at the lower level. This linking of structures accommodates an arbitrary amount of nesting and the repeated calls to the same parser accommodate chaining of indefinite length.

Our architecture naturally lends itself to the primary-backup style of replication [5], in which at any point no more than one replica is designated as *primary* and all others as *backups*. Parsers on backup hosts may wait for state updates from the primary parser for coordination purposes. To assist with that, we provide a simple broadcast operation. When all replicas are at the same point in the stream, they invoke this broadcast operation, which returns the same result on all of them: the result that the primary replica submitted. To use our interception technique with state-machine replication [7], some other method of choosing the "correct" result out of many diverging values would be required.

3. Samba interceptor

To illustrate our architecture, we discuss an interceptor for the NBT/SMB protocols used by Samba. At the lowest level, Samba messages are encapsulated by the NBT protocol. Each NBT message can contain a chain of SMB messages that all share a common header, which includes information (e.g., the session ID) common to all messages in the chain. The interceptor uses three parser functions shown in Figure 1. These functions are called on both the primary replica and the backups, and they are invoked on messages traveling in either direction.[1]

NBT_PARSER is invoked upon the receipt of every NBT message. To help higher-level parsers process multiple SMB messages that can be chained together inside an NBT message, NBT_PARSER saves the full length of the packet (nbt_length) in the state and resets the size of the message processed so far. It then returns a reference to a higher-level parser (SMB_MAIN_PARSER), a specification of 32 bytes (enough for that parser to make a decision), and a remainder consisting of the data portion of the NBT message. After everything in the remainder has been processed, NBT_PARSER is invoked again, at the boundary of the next NBT message.

To process the remainder, SMB_MAIN_PARSER is invoked with the first 32 bytes of the Samba header. SMB_MAIN_PARSER extracts the first command and its type (REQUEST or REPLY), and, based on these values, decides how much of what remains in the Samba header to pass to the next parser (SMB_PARSER). We use two separate parsers because Samba messages have a general header followed by one or more chained messages with their own mini-headers. SMB_MAIN_PARSER is invoked once per NBT message to parse the general header and SMB_PARSER is invoked once for every chained message contained inside.

SMB_PARSER resolves the application-level state differences by performing substitutions on header fields in certain messages. To do so, the primary parser sends messages to the backups using the *broadcast_state()* function. On the primary replica this function sends its argument to the backups and returns as soon as the argument is known to be stable; on the backups the function blocks until the value is available and then returns the value that the primary sent.

There are two potential sources of nondeterminism that SMB_PARSER resolves:

(1) There is a potential difference in the **random challenge** sent by the server during authentication. The challenge is sent in a NEGPROT message, at which point the parser records both the challenge generated locally and the challenge generated by the primary. When a client-generated SESSSETUPX message is later intercepted, the parser authenticates the response using the primary's challenge, and if that succeeds, it generates a valid response using the local challenge. Both *authenticator()* and *generate_response()* functions need access to the encrypted password for the user. (Since the password must be accessible to the server process, it can be made accessible to the interceptor as well.) If authentication fails then any invalid string can be returned to ensure that authentication inside Samba also fails.

(2) There is a potential difference in **random handles** chosen by the server for each open file. Given a handle, the function *find_matching_handle()* returns its match. If there is none, as happens with NTCREATEX and OPENX reply messages, the mapping is established by broadcasting the primary's handle to backups and having all replicas add the mapping to their state.

The last few lines of the pseudo-code for SMB_PARSER handle chained messages. The names of such messages end in AndX, which indicates that another message follows the current message at a certain offset from the beginning of the NBT packet. When SMB_PARSER is invoked, the command, the type, and the

[1] For clarity, we omit some details of the implementation: parsing and verification of protocol headers is delegated to functions that return an *msg* structure, which can be used both to read and modify parts of the data (i.e. the changes to an *msg* are reflected in the data upon exiting the parser); performing authentication and management of handles is also abstracted. Furthermore, we assume that the data is in the address space of the current context, while in our kernel-level implementation of parsers the data needs to be copied from user address space into kernel address space and back.

	Overhead (μsec/byte)	Overhead/0.08 (% of 100 Mbps)	Throughput Loss
Bulk SMB (0.06%)	0.0000028	000.003%	0
Small SMB (13%)	0.0005707	000.713%	0
Full Copy	0.0042402	005.301%	0
Full Scan	0.0046478	005.810%	0
Full Decryption	0.0528111	066.014%	28%
Full De- & Encryption	0.0943686	117.961%	45%

Table 1. *Interception overhead.*

offset of the next chained message are available in *msg*. Unless the command has a special value END_OF_CHAIN, which indicates that this is the last message in the chain, the name and type are remembered in the state, and the remainder is shortened to the offset. SMB_PARSER is then invoked again on the next message. Since the parser never modifies the contents of the message, it returns an *Unrestricted* spec and a NULL parser pointer.

4. Overhead of interception

The overhead of interception depends on the nature of traffic on the connection, the type of work that the interceptor has to perform, and the hardware configuration. Table 1 shows what overhead can be expected for a wide range of applications on contemporary commodity hardware (1.4-GHz Pentium III workstations with a 512-KB L2 cache and 1 GB of RAM) and a saturated 100-Mbps network link. The overhead is shown in three ways: the average delay in microseconds imposed on each byte in the stream (as measured by timers surrounding interceptor invocations), the same delay as a percentage of time it takes to transfer a byte at 100 Mbps, and the actual application-level throughput loss measured at the client.

The first two rows are for Samba interception: in one, full-sized SMB messages were sent while performing a bulk data transfer and in the other the session consisted of small SMB messages (the percentages in parenthesis show the portion of all transferred bytes that the interceptor had to extract). We expect this to be representative of the range of overheads for a typical unencrypted session, in which at most several bytes need modification in each application message.

The third row shows how much it costs to copy *all* bytes in the stream in and out of the interceptor's address space;[2] in the fourth row, a scan through the copied data was added to the copy operation. Although the imposed delay is significant in relation to the network interface speed, no throughput loss was observed because TCP buffering absorbed the added delay.

When DES encryption was added as a processing step, though, the client throughput dropped significantly, as can be seen in the last two rows. When the interceptor has to decrypt, modify, and re-encrypt the I/O stream to remain transparent (e.g. when manipulating data inside an SSL connection), the throughput drops by almost a half. Therefore, offloading the encryption algorithm to hardware would be necessary to maintain an adequate level of performance with such an application.

[2]Most of this overhead can be avoided if the interceptor can manipulate the data directly.

5. Discussion

Self-induced nondeterminism is by far the major source of malignant nondeterminism we have found in servers. Often, it was the only form of malignant nondeterminism that we observed. In our earlier work with FT-TCP [1, 9], we removed this nondeterminism by modifying the server's code. That approach is not always available, and so we developed this interception approach to have a more widely usable method.

Although we are not aware of any prior work in which nondeterminism is accommodated by manipulating the I/O streams of the application, our interceptors are conceptually similar to "middleboxes," such as firewalls or NATs, which interpret and modify network packets. In fact, it seems likely that nondeterminism interceptors would be deployed together with these devices.

We have written interceptors for several servers, including Apache, Samba, and Apple's Darwin Streaming Server. We found writing interceptors to be reasonably straightforward, and as long as re-encrypting data is not needed, the overhead of using interception is small. In addition to the random handles and challenges, as seen in Samba, we also encountered nondeterminism in the textual messages from the server (e.g. a random "fortune" printed by the server upon logging in, a report of the throughput reached by a transfer) and in the unique file names chosen by the server upon client's request (e.g. the STOU command of the FTP protocol). All of these could be easily reconciled with an interceptor.

Whether these observations are true for the majority of distributed applications remains a subject of future research. We also would like to understand better why, once self-inflicted nondeterminism is resolved, there is so little malignant nondeterminism in the network services we studied. If one can be assured that, for a wide set of services, it is indeed a rare event, then it may be best resolved by treating it as a failure: essentially, another rare form of malignant nondeterminism.

```
var state // connection-specific state

NBT_PARSER (data, size)
    msg ← parse_nbt_header(data, size);

    state.size ← 0; // portion of the Samba message seen so far
    state.nbt_length ← msg.length; // size without NBT header
    remainder ← msg.length;
    spec ← 32; // size of the first portion of the Samba header
    parser ← &SMB_MAIN_PARSER;

    return {remainder, spec, parser};


SMB_MAIN_PARSER (data, size)
    msg ← parse_main_smb_header(data, size);

    state.type ← msg.type; // REQUEST or REPLY
    state.command ← msg.command; // first command in the chain
    remainder ← state.nbt_length - size;
    // enough bytes to parse this specific command
    spec ← snap_length(msg.command, msg.type);
    parser ← &SMB_PARSER;

    return {remainder, spec, parser};


SMB_PARSER (data, size)
    // state is necessary for command and the type
    msg ← parse_smb_header(data, size, state);
    remainder ← state.nbt_length - state.size - size;

    ////////// Samba authentication //////////
    if msg.command = NEGPROT
    ∧ msg.type = REPLY then
        state.my_challenge ← msg.challenge;
        state.primary_challenge ← broadcast_state(msg.challenge);

    if msg.command = SESSSETUPX
    ∧ msg.type = REQUEST then
        if authenticator(
        state.primary_challenge, msg.response) = VALID then
            msg.response ← generate_response(
            state.my_challenge);
        else
            // this will fail to authenticate
            msg.response ← ERROR;

    ////////// Samba file handles //////////
    if msg.handle ≠ NULL
    ∧ msg.type = REPLY then
        handle ← find_matching_handle(
        msg.handle, state.handles, REPLY);

        if handle = NULL then
            // file handle seen for the first time:
            // primary sends, backups receive
            handle ← broadcast_state(msg.handle);
            state.handles ← state.handles ∪ { handle,
            msg.handle }; // remember the mapping
        msg.handle ← handle;

    if msg.handle ≠ NULL ∧ msg.type = REQUEST then
        msg.handle ← find_matching_handle(msg.handle,
        state.handles, REQUEST);

    ////////// Samba request chaining //////////
    if AndXCommand(state.command) = TRUE
    ∧ msg.command ≠ END_OF_CHAIN then
        state.type ← msg.type; // REQUEST or REPLY
        state.command ← msg.command; // next command in chain
        // shrink the remainder so this parser is called again
        remainder ← msg.offset - state.size - size;
    state.size ← state.size + remainder + size;

    return {remainder, UNRESTRICTED, NULL};
```

Figure 1. Pseudocode for Samba parsers.

References

[1] L. Alvisi, T. C. Bressoud, A. El-Khashab, K. Marzullo, and D. Zagorodnov. Wrapping server-side TCP to mask connection failures. In *Proc. IEEE INFOCOM*, pages 329–337, Anchorage, Alaska, USA, April 2001.

[2] J.F. Bartlett. A nonstop kernel. In *Proc. of the 8th Symposium on Operating Systems Principles (SOSP)*, pages 22–29, Asilomar, California, USA, December 1981.

[3] T.C. Bressoud. TFT: A software system for application-transparent fault tolerance. In *Proc. 28th Annual Intl. Symposium on Fault-Tolerant Computing (SRDS)*, pages 128–137, Munich, Germany, June 1998.

[4] T.C. Bressoud and F.B. Schneider. Hypervisor-based fault tolerance. *ACM Trans. on Computer Systems*, 14(1):80–107, 1996.

[5] N. Budhiraja, K. Marzullo, F.B. Schneider, and S. Toueg. Primary–backup protocols: Lower bounds and optimal implementations. In *Proc. 3rd IFIP Conf. on Dependable Computing for Critical Applications*, pages 187–198, Mondello, Italy, September 1992.

[6] Eric Daniel and Gwan S. Choi. TMR for off-the-shelf Unix systems. Short presentation at IEEE Intl. Symp. on Fault-Tolerant Computing (FTCS), June 1999.

[7] F.B. Schneider. Implementing fault-tolerant services using the state-machine approach: a tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[8] D.P. Siewiorek and R.S. Swarz. *Reliable Computer System Design and Evaluation*. Digital Press, Bedford, Massachusetts, USA, 1992.

[9] D. Zagorodnov, K. Marzullo, L. Alvisi, and T.C. Bressoud. Engineering fault-tolerant TCP/IP servers using FT-TCP. In *Proc. IEEE Intl. Conf. on Dependable Systems and Networks (DSN)*, pages 393–402, San Francisco, California, USA, June 2003.