TurboSockets: Democratizing Distributed Deduplication

João Salada and João Barreto
INESC-ID/Technical University Lisbon, Portugal
Email: joao.salada@ist.utl.pt and joao.barreto@ist.utl.pt

***Abstract*—Distributed deduplication is one of today's most prominent techniques for efficient data transfer of data across a network. However, leveraging a distributed application with distributed deduplication capabilities is a complex challenge, not accessible to the average programmer.**

This paper advocates that the time has come to devise general-purpose middleware abstractions that can democratize the use of state-of-art distributed deduplication techniques, even by programmers with no know-how on the field. We propose *TurboSockets*, the first middleware abstraction that aims at such a goal. The TurboSockets middleware enables unskilled programmers to establish a communication channel between two remote processes and, through that channel, exchange data streams whose content is deduplicated by state-of-the-art algorithms. Turbosockets hides all the complexity associated with the deduplication protocol away from the programmer, closely resembling traditional inter-process communication APIs.

Using a full-fledged prototype of the TurboSockets middleware, experimental results with real workloads confirm gains in performance and transferred volumes for a wide range of real workloads and scenarios.

***Keywords*-distributed deduplication; data redundancy; sockets; middleware;**

I. Introduction

We are facing a digital revolution with giant proportions. The International Data Corporation (IDC) forecasts that 35 zettabytes will be generated in 2020, while in 2011 it was 1.8 zettabytes [1]. Storing and transferring such massive amounts of data will inevitably have substantial costs.

Fortunately, a crucial opportunity to reduce such costs arises from the inherent redundancy stemming from the data we generate. More precisely, the IDC estimates that 75% of the generated data results from copies from previous data [2]. Consequently, a considerable percentage of data that we store or transfer across a network will be similar to data already stored at the target device or machine. Not surprisingly, techniques that are able to detect and eliminate such redundancy are more and more omnipresent in today's applications, enabling companies to reduce the cost from creating, capturing, managing and storing information since 2005 to 2011 [1] to one sixth.

In this paper we focus on the costs of transferring large amounts of data between processes across a network.

One of today's most prominent techniques for efficient data transfer of data across a network is *distributed deduplication* [3], possibly combined with conventional techniques such as data compression or caching. Distributed deduplication comprehends a set of techniques that try to remove redundant data from streams that are about to be transferred between a pair of remote processes, with the intent of reducing network consumption costs and/or improve network performance. The key insight is that the receiver may already hold some chunks of data that are similar to chunks in the data stream that the sender wishes to transfer. With a proper distributed deduplication protocol, such *redundant chunks* may be detected and, thus, not transmitted across the network; instead of their contents, the sender needs only to transmit lightweight references that tell the receiver where, within the contents that it holds locally, it can obtain the redundant chunks.

A number of reasons helps one understand the importance of distributed deduplication today. First, distributed deduplication has been critical in ensuring acceptable QoS in services that demand transferring large amounts of data in short periods of time, e.g. cloud storage services, despite bottlenecks in wide-area networks [4]. Second, distributed deduplication has also shown great benefits in accelerating WANs of the developing world, where bandwidth is scarce and expensive [5]. Finally, the bandwidth and battery limitations of mobile devices can often be minimized by employing distributed deduplication [6; 7]. Not surprisingly, many of the most successful IT companies employ distributed deduplication at the core of their services. Dropbox [8], Amazon and Google [9] are only two examples among many whose success depends on effective distributed deduplication.

Yet, when one shifts one's attention to the universe of lower-profile distributed applications, developed and maintained by less resourceful programmers, it becomes evident that distributed deduplication is absent from the vast majority of such applications. The reason is simple: to the best of our knowledge, there is not any easy-to-use generic middleware abstraction that allows the average programmer to leverage their distributed applications with deduplication techniques without becoming a deduplication expert.

In fact, should a programmer decide to incorporate distributed deduplication in his/her application, such a programmer is required to implement complex and error-prone protocols. This represents a relevant challenge that the average programmer will be reluctant or just unable to take on. Furthermore, the option of importing the source code of some deduplication library, implemented by others in the context of a distinct project, is usually inapplicable, as most deduplication solutions are specifically tailored to particular workloads and protocols (e.g. web, virtualization, replication), hence hampering their generic use.

This paper advocates that distributed deduplication should be democratized through a suitable general-purpose middle-

This work was supported by national funds through FCT - Fundacão para a Ciência e a Tecnologia under projects PEst-OE/EEI/LA0021/2011 and specSTM (PTDC/EIA-EIA/122785/2010)

ware abstraction for the masses. In the same way that successful middleware abstractions such as Berkeley Sockets, Remote Procedure Calls and Remote Method Invocation [10] made distributed programming available for a wide universe of average programmers, we claim that this can also be achieved with distributed deduplication.

This paper proposes *TurboSockets*, the first middleware abstraction that aims at democratizing the use of distributed deduplication protocols. The TurboSockets middleware allows for the average programmer to establish a communication channel between two remote processes and, through that channel, exchange data streams whose content is deduplicated by state-of-the-art algorithms. Turbosockets hides all the complexity associated with the deduplication protocol away from the programmer. Thus, the programmer is neither required to have any prior-knowledge of deduplication nor to be an expert programmer to use distributed deduplication in his systems. The Turbosockets API is designed to closely resemble the BSD Sockets [10], which are familiar to many programmers.

In order to be useful for most applications, the TurboSockets middleware needs to meet some fundamental requirements:

1) As the state of the art in the fields evolves, Turbosockets must be extensible so as to be upgraded with new deduplication algorithms.
2) Since different applications and workloads call for different deduplication solutions [11; 12; 3], Turbosockets must support multiple algorithms and combinations thereof, and enable programmers and applications to easily fine-tune the algorithms and parameters that are most suitable for each particular application and workload.
3) Since two processes may have individually set distinct algorithms and parameters, Turbosockets must be able to cope with such heterogeneity.

The above requirements incur significant challenges that our solution must address. Overcoming such challenges while retaining the desired abstraction simplicity and, simultaneously, achieving good performance is far from trivial.

Our contribution, Turbosockets, accomplishes this by relying on a tunable extensible distributed deduplication protocol, encapsulated inside a socket-like communication channel abstraction. Our middleware constitures a generic framework can be extended by any expert programmer by plugging in new algorithms in a modular fashion. Hence, TurboSockets is also a valuable tool for the research community, as it provides researchers with a way of effortlessly test new algorithms and optimizations in the TurboSockets framework, instead of building new protocol prototypes and applications from scratch.

We have implemented a full-fledged prototype of the TurboSockets middleware. The library is publicly available for any programmer to use using one of 19 possible programming languages (including C, Java, Python, PHP, Ruby, C#, and CLISP).[1]

We have evaluated the library implementation using a file transfer application that we built relying on TurboSockets. Our experimental results show that, on different scenarios with real workloads:

1) TurboSockets are able to significantly transfer less bytes across the network, when compared to a baseline solution using (undeduplicated) Berkeley Sockets (up to 81% reduction).
2) In limited bandwidth scenarios, TurboSockets prove to be substantially faster (up to 80% with 3 Mbps)than Berkeley Sockets.

The remainder of the paper is organized as follows. Section 2 surveys related work on distributed deduplication. Section 3 introduces our solution. Section 4 evaluates. Finally, Section 5 draws conclusions and discusses future work.

II. Related Work

Much work, with the objective of decreasing bandwidth usage or latency of data transfers, has been developed in the most diverse areas. Distributed deduplication has been successfully applied in project repositories, such as, Git [2] or CVS [13]. Several works on the Web [14; 7], demonstrated that distributed deduplication reduces both latency and traffic of http-based systems. Wide-area networks in developing countries were accelerated employing distributed deduplication [5]. Proposals in remote file systems [15], multi-user file sharing systems [8] and replication systems [16; 17] also showed great improvements with distributed deduplication.

A. Data Division

The first phase of distributed deduplication is the redundancy detection. Given a stream to deduplicate at the sending node (sender), one must first identify which chunks of it are already present at the receiving node (receiver). A stream is an array of bytes, while a chunk is a sub-part of the stream. A chunk may the same size as the the stream or smaller. The first step to deduplicate is to divide the stream into chunks.

The granularity at which data division occurs assumes a preponderant role in the overall deduplication performance. For any distributed deduplication solution, the lower the granularity the better is redundancy detection, but at the cost of an increase on metadata footprint [17; 12; 3].

The coarsest granularity does not divide the stream, as it tries to find an identical stream to the one being deduplicated. We call it *whole-stream* division (WSD) and it has been widely employed [12; 18; 3; 8]. Meyer et. al. performed a deduplication evaluation on 857 Windows desktops and found that whole-stream achieved three-quarters of the space savings obtained by the finest-granularity[12]. On the other hand, Policroniades, C. [11] was only able to get 5% of space savings in data-sets with no evident correlation, and where finer granularities reached gains up to 20%.

Chunks may also be divided into non-overlapping fixed-sized chunks. Rsync [19], a tool which identifies redundant parts of files stored in different machines, was one of the first deduplication solutions to use *fixed-sized division* (FSD) in literature. Other deduplication works employing FSD are TAPER [17] or OS Streaming Cache [20]. FSD does not deal well with modifications [11]. However, it presents interesting advantages, such as, better space and computacional

[1]The TurboSockets library source code is publicly available at https://bitbucket.org/jsalada/turbosockets

[2]http://git-scm.com/

efficiency when compared to variable-size division [21], and performs better than whole-stream division [12].

In order to overcome the propagation of chunk modifications that imply shifting data (insertions or deletions of data) into the further chunks, *variable-sized division* was introduced. This is accomplished at the expense of more computational and storage complexity [3]. Since physical storage blocks are fixed-sized, variable-size chunks will not fit exactly in one physical block, adding additional overhead to overcome the situation [22].

Content-Defined Chunking (CDC) [15] is the most common technique to perform variable-size division technique, which have been optimized in a number of ways [22; 23; 24]. CDC examines every overlapping fixed-sized chunks of a data stream, using a *sliding window*. Whenever the *window* satisfies a certain condition, a new *breakpoint* is found. Breakpoints delimit the boundary regions which will divide the data stream into new chunks. They are selected using an efficient hash function, such as *Rabin fingerprints* [25]. Minimum and maximum size restrictions on chunk size ensure pathological cases do not corrupt the deduplication protocol.

CDC shows a consistently better effectiveness when compared to fixed-size division or whole-stream [11; 12; 3], although being the most expensive in terms total deduplication time and CPU usage [3].

B. Identifier Generation

After identifying chunk boundaries, their identifiers are generated. There are two prominent groups of identifier types, *identical detection identifiers* and *similar detection identifiers*.

The identifiers that detect identical chunks are hash values of chunks, generated by an hash function. A hash function *H* accepts any chunk *d* as input and returns a hash *h*, meaning *H(d)=h*. The hash *h* is the chunk identifier. In the context of deduplication, the first important property of the hash functions is they must output a hash where $Size(h)$ is much lesser than $Size(d)$. The second property must guarantee that the probability of two different chunks, d_1 and d_2, generate identical hash-values is very low. A collision occurs when $H(d_1) = H(d_2) = h$, resulting in possible data corruption.

The most widely used hashing techniques, in deduplication systems, are cryptographic hash functions (CHF). The CHF, *MD5* [26], was used in *Presidio* [18]. *SHA-1* [27], a CHF of160 bits, is also very popular in deduplication, being used in *LBFS* [15], Presidio [18]. Other CHF are present in deduplication works, such as the *SHA-256* [27] in OS Streaming [20].

Non-cryptographic hash functions (NCHF) have also been employed with success in deduplication systems. The NCHF Super-Hash[3] generated identifiers for a memory deduplication system [28] and MurmurHash[4] for a mobile web application [7].

The collision probability, P_C, depends directly on the number of bits of the hash function and the total amount of data that the deduplication system will process during its entire lifetime. *Uou et. al.* [18] present a derivation of the original P_C equation [29; 30; 31; 18], which allows to dimension the system's P_C, as a function of the $Size(h) = b$ and the expected total amount of data's magnitude order q, that is going to be deduplicated:

$$P_C(b, q) \approx 2^{2q-(b+1)} \quad (1)$$

An adequate collision dimensioning is important not only to prevent high collision rates, but also because many designs do not need the strongest hash function to keep collision rates at acceptable values. Black [31] criticizes the general standard of using CHF in deduplication, when there is no adversaries, stating that systems enter in overkill either by relying on strong hashes (many bits), either by selecting CHF.

When deduplication is based on *delta-compression*, one chunk very similar to the one being deduplicated must be firstly identified. Different types of identifiers may be created for similarity detection. *Sketches* are a set of identifiers from different parts of a chunk. If two chunks hold sketches with many identifiers in common, then they are considered similar. This scheme was applied in former delta-enconding proposals [25; 32; 33]. *Super-sketches* [34; 18] are summaries of sketches. Sketch's identifiers are coalesced into groups of more than one, and hashed into a super-identifier. Finally, *similarity hashes* or simhashes are created by an hash function, where two similar chunks will generate similar hash values in terms of Hamming Distance [35]. Similar chunks have simhashes with many common bits.

C. Redundancy Detection

As soon as chunk identifiers of the stream to send are computed, the deduplication protocol exchanges such identifiers and tries to determine which of such chunks are already present at the receiver process. This problem is not trivial to solve in a scalable and efficient way. Hash Challenges try to minimize the metadata exchanged by transmitting only a part of each identifier first [36]. Some solutions index chunks at the receiver by their identifiers directly into an *in-memory* hash database, namely a simple *hash-table* implementation [16] or a more sophisticated, but still very simple, software [17; 15]. More complex designs have been proposed to cope with the scalability issue. DDFS [21] and Presidio [18] implement the storage of the identifiers index in disk, while memory works as a cache.

More improvements to these features have been proposed. MAD2 [37] uses the same base concepts of DDFS and distributes deduplication by several nodes to increase scalability and throughput. Extreme-Binning [38] stores identifiers and chunks in a common container if they belong to the same or similar stream. Another suggested scheme [39] utilizes *sparse indexing* to explore disk locality with the use of containers and a sparse-index. Shilane et al. [40] transposes the DDFS system to a distributed deduplication solution by adding super-sketches of chunks to the metadata stored in *containers*, so that delta compression may be performed over chunks where compare-by-hash failed to detect redundancy.

III. TurboSockets

This section describes the TurboSockets API in the perspective of the programmer that may use this work in his applications. Then, we cover all the architecture of TurboSockets, which is illustrated in Figure 1.

[3] redshift.sourceforge.net/superhash/

[4] http://code.google.com/p/smhasher/wiki/MurmurHash

A. TurboSockets API

The current TurboSockets API supports TCP TurboSockets, offering identical primitives to the BSD Sockets API, such as, *socket, bind, listen, accept, connect, recv and send* (see Table I).

Comparing to BSD Sockets, the only additional elements in TurboSockets API are the primitives related to the *Environment* (Table I). The Environment provides TurboSockets with access to the chunk database and automatic internal memory management. In order to execute any primitive at TurboSocket instances, an Environment must exist. An Environment instance may be associated to more than one TurboSocket instance.

TurboSockets API is virtually available to programmers of any language. We implemented the API using *SWIG*[5], an interface compiler that generates code wrappers for nineteen program languages, such as, Java, Python, PHP, Ruby, C#,CLISP, etc. This assures our proposal adapts to the programmer and not the opposite.

Table I
THE TURBOSOCKETS API IN C

TurboSocket API
*int ts_socket(ts_env_t *env,int family,int type,int proto,,ts_socket_t **tsock);*
*int ts_socket_set_addr(ts_env_t *env,ts_socket_t *tsock,int hostname,int family,int dedup_port,intundedup_port,int flags);*
*int ts_socket_bind(ts_env_t *env,ts_socket_t *tsock);*
*int ts_socket_listen(ts_env_t *env,ts_socket_t *tsock,int backlog);*
*int ts_socket_accept(ts_env_t *env,ts_socket_t *tsock,ts_socket_t *new_conn);*
*int ts_socket_connect(ts_env_t *env,ts_socket_t *tsock);*
*int ts_socket_send(ts_env_t *env ,ts_socket_t *tsock, char *buff, ts_stream_sz_t *buff_sz);*
*int ts_socket_recv(ts_env_t *env,ts_socket_t *tsock,ts_stream_sz_t *buff_sz);*
*int ts_socket_destroy(ts_env_t *env,ts_socket_t *tsock);*
Environment API
*int ts_env_create(ts_env_t **env);*
*int ts_env_destroy(ts_env_t *env);*
*void ts_env_set_db_filename(const char *name);*
*void ts_env_log_filename(const char *name);*

B. TurboSockets Architecture Overview

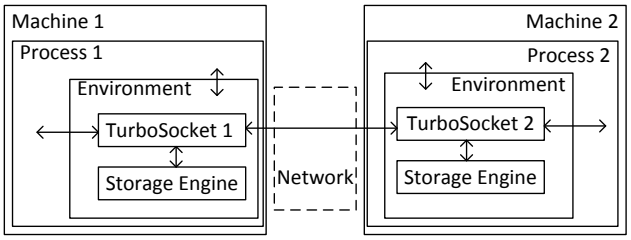

Figure 1. The high-level architecture of two TurboSocket pairs

TurboSockets aim at reducing the bandwidth usage of applications that transmit data across remote nodes. To achieve that, TurboSockets offer bidirectional channels of communication where different processes may exchange data streams, while transparently, state-of-art deduplication techniques detect and remove duplicate data from the exchanged streams (Figure 1).

Using the *send call* of TurboSockets API, any process may submit data for deduplication and transmission. In such a situation, we define as *sender* the TurboSocket end calling the *send* primitive, and *receiver* the other TurboSocket end that will receive the data. All the ends of a TurboSockets communication channel may assume both behaviours, as sender and receiver.

[5]http://www.swig.org

TurboSockets process data as byte arrays, which we call streams. The Chunk Division module (CD) at the sender initiates the deduplication process, by dividing the submitted streams into chunks. The objective is to identify which chunks within the stream already exist at the receiver, so the sender will not transmit them. Following the CD phase, the Identifiers Generation module (IG) computes hash-based identifiers of the new chunks. The chunk identifiers enable the detection of duplicates, during the Redundancy Detection (RD) phase, where the sender transmits new chunk identifiers to the receiver which compares them against the identifiers of the chunks locally present at receiver. Then, the receiver communicates the sender which stream chunks are *redundant*. A stream chunk is redundant when it is a duplicate of other chunk at the receiver. Once the sender knows which chunks are redundant it proceeds to the Redundancy Elimination phase (RE) of the stream, transmitting only the chunks data that the RD was not able to find a duplicate at the receiver. Then, the Stream Reconstruction module (SR) at the receiver, reconstructs the stream with the chunks data transmitted by the receiver plus the redundant chunks data retrieved by the SE.

One process might run several instances of TurboSockets (e.g. one webserver could have one TurboSocket per client), which might be aggregated within a single *Environment*. An Environment provides all its TurboSocket members with access to the same *chunk database*. The chunk database allows to look up the stream chunk identifiers for redundancy within a large set of chunks, and stores the chunks' data to enable the reconstruction of redundant chunks locally at the receiver. The *Storage Engine* (SE) manages the chunk database. The association of several TurboSockets into sharing the same SE leads to: 1) Storage space savings, as the SE stores chunks appearing in more than a TurboSocket only once; 2) Increased RD ability, as one TurboSocket can access all the chunks that other TurboSocket instances sharing the same Environment received.

TurboSockets support an *hierarchic multi-level deduplication* approach. Each deduplication level tries to find redundant chunks with different deduplication algorithms and parameters. Levels are hierarchic, meaning the top level first deduplicates all the data arriving for transmission, then the resulting non-redundant chunks feed the immediate lower level. When no more levels are available, the sender transmits the remaining non-redundant chunks data to the receiver.

TurboSockets are *highly configurable* and two instances with different deduplication configurations can deduplicate and exchange data between each other, despite their differences. An *handshake protocol* is performed immediately after two TurboSockets establish a new connection. During the handshake protocol they exchange their respective configurations and reach an agreement about the configuration to employ. To enrich configurations TurboSockets provide a generic framework of deduplication, with state-of-art algorithms built-in. Expert programmers may easily plug in new algorithms in a modular fashion.

BSD Sockets support TurboSockets in establishing all the message exchanges across remote TurboSocket instances.

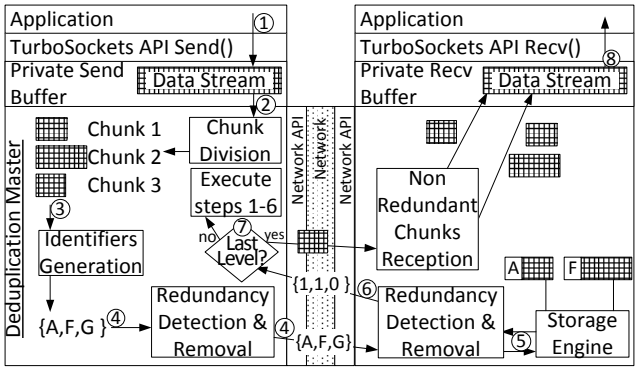

Figure 2. The Distributed Deduplication Protocol of the TurboSockets API

C. TurboSockets - Distributed Deduplication Protocol

The *TurboSockets-Distributed Deduplication Protocol* (TS-DDP) performs the distributed deduplication of data streams across remote TurboSocket nodes. The protocol is illustrated in the Figure 2. The TS-DDP comprises the *Deduplication Master* in charge of the deduplication coordination, plus the components that effectively deduplicate the stream, *Chunk Division, Identifier Generation, Redundancy Detection*, *Redundancy Elimination* and the *Stream Reconstruction* (SR) module. We describe each one next:

1) Deduplication Master: The *Deduplication Master* (DM) coordinates the TS-DDP. In order to explain the algorithm followed by the DM (Figure 2) let us assume that two remote points, the sender node and the receiver node, are running an appplication that exchanges data through a TurboSockets channel. The example shows unidirectional communication for the sake of simplicity, however TurboSockets offer bidirectional data transmission. Next we explain the algorithm.

1) In order to send data, the application calls the *send* function, which copies the data array to the *Private Send Buffer* (PSB), where data stays until the DM consumes it and initiates the TS-DDP. The data that the DM consumes from the PSB and delivers to deduplication forms a stream.
2) Each stream enters the CD module. The CD divides the the stream into chunks according to the division algorithm configuration that is set for the current deduplication level.
3) The IG module generates a hash-based identifier for each chunk, according to the configured Identifier Generation Algorithm.
4) The Redundancy Detection and Removal module (RDR), which comprises the RD and RE phases, packs all the newly generated chunk identifiers into a single message, along with the respective chunk offset (within the stream) and size. The RDR transmits the message, called *request message*, to the receiver.
5) At the receiver, the RDR receives and processes the request message. The RDR performs queries to the SE with each identifier present in the request message. If the identifier is present in the SE, then it is considered redundant, otherwise it is considered non-redundant. The SE returns all the chunk *recipes* associated with the redundant identifiers. A recipe is the data structure that enables the chunk reconstruction locally and contains for each chunk: the identifier, the chunk offset and the chunk size . The chunk sizes of redundant identifiers are compared against the size present in the recipe. If sizes do not match, a collision is detected and the identifier is considered non-redundant. Otherwise, chunk reconstruction takes place with assistance of the recipe. The DM stores the reconstructed chunk data within the *Private Receiver Buffer* (PRB) where it remains unavailable for the application to consume, until the complete data stream is present in the PRB.
6) Once the RDR knows which chunks are redundant, it fills and transmits the *reply message*. The reply message encodes the redundancy of each chunk present in the request message into a *bitmap*. The bitmap assigns one bit to each identifier of the request-message. If the bit is set, then the identifier is redundant. If the bit is not set, the identifier is not redundant. The bit position of each identifier is identical to the position of the same identifier in the request message. This policy allows the sender to identify what is the identifier encoded by a certain bit, without the need of transmitting the identifier again. In the example of Figure 2, the request-message contained the identifiers $\{$A,F,G$\}$, and being only G non-redundant, the bitmap looks like $\{1,1,0\}$.
7) One of the following steps occur:
   a) If there are more levels of deduplication to perform, the DM saves the non-redundant chunks from the reply message and forwards them to the next level of deduplication, where the DM executes the steps 2-7 again, with the configuration of the next level of deduplication. In the case where the bitmap declares that all chunks are redundant, the stream deduplication is finished.
   b) If no more deduplication levels are available, the DM packs all data from the chunks marked as non-redundant in the bitmap, into a message which we call *data-message*. The DM transmits the data-message to the receiver node.
8) When the distributed deduplication of a stream is finished, the receiver node already holds a copy of the original stream at the PRB, which may now make the stream available for consumption by the API *recv* call.

2) Chunk Division Module: We designed TurboSockets to support the inclusion of different division algorithms (Section III-C2) in a very simple and modular manner. New additions just require to implement the interface presented in the Table II. The *serialize* and *deserialize* functions enable the exchange of configurations between TurboSockets instances, during the initial handshake protocol. TurboSockets support several *data division* algorithms, including WSD, FSD and CDC as built-in options. The programmer has complete freedom to configure algorithms with all possible parameter configurations (e.g granularity, maximum chunk size).

Table II
DIVISION ALGORITHM API IN C

*int div_alg_next_chunk(ts_env_t *env,ts_segment_t *seg,ts_div_info_t *info,ts_segment_t **chk)*
*ts_div_alg_t *div_alg_div_alg_init()*
*char *div_alg_serialize()*
*ts_div_alg_t *div_alg_deserialize(ts_div_alg_t *alg_serialized)*

3) Identifiers Generation Module: The IG module offers modular Identifier Generation Algorithms (section II-B), since any expert programmer might plug in new Identifier Generation Algorithms by implementing the interface of Table III. TurboSockets do not need to know what the generated identifier represent, as it also the Identifier Generation Algorithm that performs the RD, with the function *is_chunk_redundant()*. This design allows to easily integrate IGAs other than compare-by-hash (e.g. super-sketches, simhashes). The TurboSockets delivers MurmurHash3 128 bits as the built-in Identifier Generation Algorithm.

Table III
IDENTIFIER GENERATION ALGORITHM API IN C

*int id_alg_generate_gen(ts_env_t *env,ts_segment_t *chk, ts_identifier_t *id)*
*int id_alg_is_chunk_redudant(ts_env_t *env,ts_segment_t chk,ts_identifier_t *id)*
size_t id_alg_size()
*id_alg * id_alg_init()*
*char *id_alg_serialize()*
*ts_id_gen_t id_alg_deserialize(char *alg_serialized)*

IV. EVALUATION

In this section we evaluate our implementation of the TurboSockets middleware. More precisely, we intend to answer two decisive questions: *What is the amount of data that an application is able to reduce when transferring data a across a pair of TurboSockets?*, and *What is the overhead introduced by TurboSockets?*.

A. Methodology

We implemented the prototype *TS-Proto*, which includes two modules: the *TS-Proto-Lib* and the *TS-Proto-Exp*. The TS-Proto-Lib, implemented in C, features the TurboSockets middleware that we described in this document. The TS-Proto-Exp, implemented in C and Python, enables experiments over the TS-Proto-Lib by including the following functionality: 1) Transfers workloads from one site (sender) to the other (receiver) using the TS-Proto-Lib to perform the data transferences. 2) Measures the experimental results, such as, the number of redundant bytes detected, total transferred volume and execution time.

Two machines interconnected by a network performed the experiments. Each machine runs the TS-Proto. Both machines are *Large instances* of Amazon's Elastic Compute Cloud (Amazon EC2)[6], and present the following equivalent setup: Linux Ubuntu 12.04 64 bits, 7.5 GiB of memory, two cores at 2GHz of an Intel Xeon CPU E5-2650. In order to have reliable values about the network available bandwidth, we first benchmarked the network utilizing the iperf tool[7]. The method consisted of running iperf during 10 s for 50 times, with an interval of 10 seconds between each iperf run. The average available bandwidth obtained was 860.48 Mbps (+/- 3.17%).

On the second experiment, we used a class base queue to emulate different bandwidths. The objective is to experiment

[6]http://aws.amazon.com/ec2/

[7]http://sourceforge.net/projects/iperf/

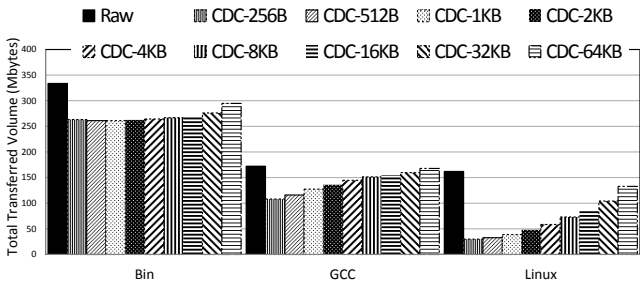

Figure 3. Total data transferred by TurboSockets pair

the behaviour of TS-Proto-Lib in relevant network technologies, namely, the IEEE 802.11n (30 Mbps), 802.11 b (11 Mbps) and Bluetooth 2.0 (3 Mbps). Again we benchmarked the network for each bandwidth emulated, resorting to the same method that was used to benchmark the network, previously. We obtained the following bandwidths: Bluetooth - 2.875 Mbps (+/- 1%); 802.11b - 11.212Mbps (+/- 0.28%); 802.11n - 30.212 Mbps (+/-0.09%).

All experiments were based on two versions of a set of files. The receiver is always in possession of the early version, and we evaluate the transmission performance of the second version. The workloads are identical to the ones used in a number of relevant papers in deduplication [15; 17; 16]. The first group of workloads are software development sources, from real-world projects, namely the Linux kernel (2.4.22 and 2.4.26) and the gcc compiler (20.1 and 20.7). The second group of workloads comprehends binaries from an operating system, Linux-Ubuntu (5.06 32-bits and 7.10 64 bits), which include the contents of the */usr/bin* directory trees. Workloads are called from now on linux, gcc and bin.

In the first experiment we wanted to measure the amount of transferred data that the TS-Proto-Lib can save, by configuring the TS-Proto-Lib with the CDC division algorithm and the MurmurHash3 128 bits. We measured the amount of data transferred for several CDC granularities. In the second experiment, to assess the efficiency performance of the TS-Proto-Lib, we measured the transfer time experimented across a TurboSockets pair, in several bandwidth contexts. In both experiments, the results obtained with bare (undeduplicated) BSD Sockets are also presented for reference.

B. Transferred Volume

In this experiment, the TS-Proto-Exp transferred all the workloads gcc, linux and bin, from one machine (sender) to the other (receiver). We configured the TS-Proto-Lib pair with several CDC granularities. Minimum and maximum chunk sizes would sit 20 KB below and above the CDC granularity. Since granularities cannot have negative values, the minimum threshold for minimum chunk granularity was set to 256 B. Figure 3 shows the result of this experimentation, where *raw* corresponds to the BSD Socket results. Comparing to the BSD Socket, TS-Proto-Lib saved bandwidth in all workloads. Linux workload experimented bandwidth savings up to 81%, gcc workload showed 37% of savings and finally the bins had a bandwidth usage reduction of 22%. We also can observe that the amount of redundancy detected decreases as we increase chunk granularity.

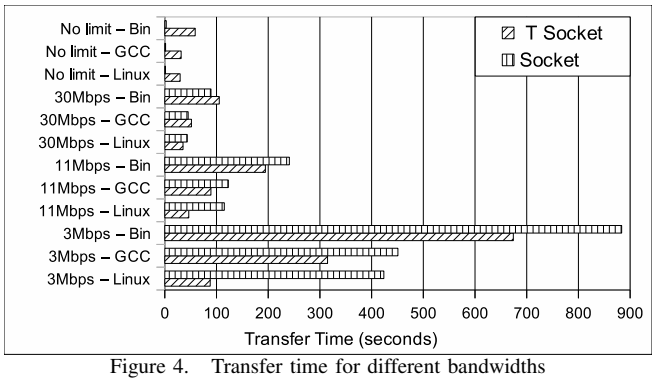

Figure 4. Transfer time for different bandwidths

C. Performance

Once we measured the direct benefits of the TS-Proto-Lib in reducing the bandwidth usage, it was important to estimate how much delay was being introduced to the data transmissions. Therefore, we set the TS-Proto-Lib to a fixed configuration of CDC with chunks of 2KB, ran all workloads, and compared the time taken to finalize each transference against the time taken by a BSD Socket at the same bandwidth. We tested several bandwidths in order emulate different network contexts.

Figure 4 presents the obtained results. BSD Sockets (*Socket* in Figure 4) can only clearly outperform the TS-Proto-Lib (*T Socket* in Figure 4) when there is not any bandwidth constriction (*No Limit* in Figure 4) in the network with 860 Mbps. These results are very interesting, as they totally expose the overhead introduced by the TS-Proto-Lib, inherent to the computational complexity of deduplication (e.g. additional abstraction layers, hashes computation, SE accesses for chunk look-ups), plus the additional RTT that distributed deduplication requires.

For 30 Mbps (IEEE 801.11n) the TS-Proto-Lib took less 18% of time to complete the transfer of linux workloads, while for gcc and bins workloads BSD sockets were faster 15% in average. For 11 Mbps (IEEE 801.11b) the TS-Proto-Lib was always faster, taking from 20% to 60% less time. In a 3Mbps bandwidth the TS-Proto-Lib was 25% to 80% faster than BSD Socket.

D. Discussion

Despite having a clear overhead when working on a network with almost unlimited resources, our TurboSockets proposal emulated by the TS-Proto-Lib middleware is consistently faster (up to 80%) when we introduce real-world bandwidth constraints. The large bandwidth savings (up to 80%) that TurboSockets obtained in the Transferred Volume experiment (Section IV-B) lead to the lower transfer times in the Performance experiment (Section IV-C). We also argue that even with worse performance than BSD Sockets, under almost unlimited bandwidths, TurboSockets may still offer considerable advantages, in contexts where large amounts of bandwidth are available, but the price of each MB transferred is very expensive (e.g. Mobile 4G).

V. Conclusions

The time has come to devise general-purpose middleware abstractions that can democratize the use of state-of-art distributed deduplication techniques to average programmers. We propose *TurboSockets*, the first middleware abstraction that enables unskilled programmers to establish a communication channel between two remote processes and, through that channel, exchange data streams whose content is deduplicated by state-of-the-art algorithms. Turbosockets hides all the complexity associated with the deduplication protocol away from the programmer, closely resembling traditional inter-process communication APIs.

Using a full-fledged prototype of the TurboSockets middleware, experimental results with real workloads confirm gains in performance and transferred volumes for a wide range of real workloads and scenarios.

As future work, we intend to adapt comon remote invocation middleware such as Remote Procedure Call and Remote Method Invocation middleware to implicitly support distributed deduplication by relying on TurboSockets. Furthermore, we plan to work on automatic learning mechanisms that can adaptively adjust each TurboSocket's configuration in order to optimize it to the most recent data sets and communication patterns that have been observed on that TurboSocket, thereby relieving the programmer from such details.

References

[1] J. Gantz and D. Reinsel, "Extracting Value from Chaos State of the Universe," *IDC (International Data Corporation) June*, 2011.

[2] J. F. Gantz and Reinsel, "The Expanding Digital Universe: A Forecast of Worldwide Information Growth Through 2010," IDC, An IDC White Paper - sponsored by EMC, Mar. 2007.

[3] N. Mandagere, P. Zhou, and M. A. Smith, "Demystifying Data Deduplication," in *Proceedings of the ACM/IFIP/USENIX Middleware '08 Conference Companion*, 2008, pp. 12–17.

[4] Y. Tan, H. Jiang, D. Feng, L. Tian, and Z. Yan, "CABdedupe: A Causality-Based Deduplication Performance Booster for Cloud Backup Services," in *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*. IEEE, May 2011, pp. 1266–1277.

[5] S. Ihm and K. Park, "Wide-area network acceleration for the developing world," in *Proceedings of the 2010 USENIX conference*, 2010.

[6] A. Rice and S. Hay, "Measuring mobile phone energy consumption for 802.11 wireless networking," *Pervasive Mob. Comput.*, vol. 6, no. 6, pp. 593–606, Dec. 2010.

[7] R. Filipe and J. a. Barreto, "End-to-End Data Deduplication for the Mobile Web," in *2011 IEEE 10th International Symposium on Network Computing and Applications*, no. i. IEEE, Aug. 2011, pp. 334–337.

[8] J. P. Ramos, L. Veiga, and P. Ferreira, "vfcBOX : Multi-User Consistent File Sharing," in *Proceedings of the 9th International Workshop on Middleware for Grids, Clouds and e-Science*, 2011.

[9] C. Kim, K.-W. Park, and K. H. Park, "Ghost: Gpu-offloaded high performance storage i/o deduplication for primary storage system," in *Proceedings of the 2012 International Workshop on Programming Models and Applications for Multicores and Manycores*, ser. PMAM '12. New York, NY, USA: ACM, 2012, pp. 17–26.

[10] G. F. Coulouris and J. Dollimore, *Distributed systems: concepts and design*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1988.

[11] C. Policroniades and I. Pratt, "Alternatives for Detecting Redundancy in Storage Systems Data," in *Proceedings of the annual conference on USENIX Annual Technical Conference*, Boston, MA, 2004, pp. 6–6.

[12] D. T. Meyer and W. J. W. Bolosky, "A study of practical deduplication," *ACM Transactions on Storage*, vol. 7, no. 4, p. 14, Jan. 2012.

[13] B. Berliner, "CVS II : Parallelizing Software Dev elopment," in *Proceedings of the Winter 1990 USENIX Conference*, 1990, pp. 341–352.

[14] M. C. Chan and T. Y. C. Woo, "Cache-based Compaction : A New Technique for Optimizing Web Transfer," in *Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies*, vol. 00, no. 1, 1999, pp. 117–125.

[15] A. Muthitacharoen, B. Chen, and D. Mazi, "A low-bandwidth network file system," in *Proceedings of the eighteenth ACM symposium on Operating systems principles*. New York, New York, USA: ACM, 2001, pp. 174–187.

[16] J. Barreto and P. Ferreira, "Efficient Locally Trackable Deduplication in Replicated Systems," in *Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware*, 2009, pp. 103–122.

[17] N. Jain, M. Dahlin, and R. Tewari, "TAPER : Tiered Approach for Eliminating Redundancy in Replica," in *4th USENIX Conference on File and Storage Technologies*, San Francisco, CA, USA, 2005, pp. 281–294.

[18] L. L. You, K. T. Pollack, D. D. E. Long, and K. Gopinath, "Presidio," *ACM Transactions on Storage*, vol. 7, no. 2, pp. 1–60, Jul. 2011.

[19] A. Tridgell, "The rsync algorithm," *Joint Computer Science Technical Report Series*, no. June, 1996.

[20] L. Garces-Erice and S. Rooney, "Scaling OS Streaming through Minimizing Cache Redundancy," in *1st International Conference on Distributed Computing Systems Workshops*. IEEE, Jun. 2011, pp. 47–53.

[21] B. Zhu, K. Li, and H. Patterson, "Avoiding the Disk Bottleneck in the Data Domain Deduplication File System Challenges and Observations," *FAST Conference*, pp. 269–282, 2008.

[22] D. Bobbarjung and S. Jagannathan, "Improving duplicate elimination in storage systems," *Transactions on Storage*, vol. V, no. July, pp. 1–23, 2006.

[23] G. Lu, Y. Jin, and D. H. Du, "Frequency Based Chunking for Data De-Duplication," in *2010 IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*. IEEE, Aug. 2010, pp. 287–296.

[24] B. Romaski, . Heldt, W. Kilian, and K. Lichota, "Anchor-driven subchunk deduplication," in *Proceedings of the 4th Annual International Conference on Systems and Storage*, New York, New York, USA, 2011.

[25] M. Rabin, "Fingerprinting by Random Polynimials. Technical Report TR-15-81," Center for Research in Computing Technology, Harvard University, Tech. Rep., 1981.

[26] R. Rivest, "The md5 message-digest algorithm," United States, 1992.

[27] USA Security Agency, "Secure hash standard," National Institute of Standards and Technology, Tech. Rep., 2002.

[28] L. Xia and P. A. Dinda, "A case for tracking and exploiting inter-node and intra-node memory content sharing in virtualized large-scale parallel systems," in *Proceedings of the 6th international workshop on Virtualization Technologies in Distributed Computing Date*, ser. VTDC '12. New York, NY, USA: ACM, 2012, pp. 11–18.

[29] M. J. Wiener, "Bounds on Birthday Attack Times," in *IACR Eprint archive*. International Association for Cryptologic Research, 2005.

[30] D. Stinson, *Cryptography: Theory and Practice*, first edit ed., K. H. Rosen, C. Colbourn, J. Gross, and A. Odlyzko, Eds. CRC Press Inc, 1995.

[31] J. Black, "Compare-by-Hash: A Reasoned Analysis," in *2006 USENIX Annual Technical Conference*, 2006, pp. 85–90.

[32] A. Broder, "Some applications of Rabin's fingerprinting method," *II: Methods in Communications, Security, and*, 1993.

[33] U. Manber, "Finding Similar Files in a Large File System," in *Proceedings of the USENIX Winter 1994 Technical Conference*, no. October, 1994, pp. 2–2.

[34] A. Broder, "Identifying and Filtering Near-Duplicate Documents," in *Proceedings of the 11th Annual Symposium on Combinatorial Pattern Matching*, 2000, pp. 1–10.

[35] M. S. Charikar, "Similarity Estimation Techniques from Rounding Algorithms," in *Proceedings of 34th ACM Symposium on Theory of Computing*, 2002, pp. 380–388.

[36] J. Barreto, L. Veiga, and P. Ferreira, "Hash challenges: Stretching the limits of compare-by-hash in distributed data deduplication," *Information Processing Letters*, vol. 112, no. 10, pp. 380–385, May 2012.

[37] J. Wei, H. Jiang, K. Zhou, and D. Feng, "MAD2: A scalable high-throughput exact deduplication approach for network backup services," in *IEEE 26th Symposium on Mass Storage Systems and Technologies*, 2010, pp. 1–14.

[38] D. Bhagwat, K. Eshghi, D. Long, and M. Lillibridge, "Extreme Binning: Scalable, parallel deduplication for chunk-based file backup," *2009 IEEE International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems*, pp. 1–9, Sep. 2009.

[39] M. Lillibridge, K. Eshghi, and D. Bhagwat, "Sparse indexing: large scale, inline deduplication using sampling and locality," in *Proccedings of the 7th conference on File and storage technologies*, 2009, pp. 111–123.

[40] P. Shilane, M. Huang, and G. Wallace, "WAN optimized replication of backup datasets using stream-informed delta compression," in *12th USENIX Conference on File and Storage Technologies*, 2012.