Abstract

In this paper, we show that it is possible to increase the message throughput of a large-scale industrial system by selectively compress messages. The demand for new high-performance message processing systems conflicts with the cost effectiveness of legacy systems. The result is often a mixed environment with several concurrent system generations. Such a mixed environment does not allow a complete replacement of the communication backbone to provide the increased messaging performance. Thus, performance-enhancing software solutions are highly attractive. Our contribution is 1) an online compression mechanism that automatically selects the most appropriate compression algorithm to minimize the message round trip time; 2) a compression overload mechanism that ensures ample resources for other processes sharing the same CPU. We have integrated 11 well-known compression algorithms/configurations and tested them with production node traffic. In our target system, automatic message compression results in a 9.6% reduction of message round trip time. The selection procedure is fully automatic and does not require any manual intervention. The automatic behavior makes it particularly suitable for large systems where it is difficult to predict future system behavior.

Keywords: Automatic compression; Message compression; Feedback control; Performance prediction; Network performance; Mobile systems.

1. Introduction

There is a great demand for high-performance communication in today’s industry, both within and between systems. We are in the midst of the evolution into multicore CPUs which causes the computational capacity [1, 2] to grow quicker than the available communication bandwidth [3]. Large-scale industrial systems [4] have additional problem areas, for example, the large and very expensive legacy of already installed systems. It is not always economically feasible to replace current systems with newer ones just because they can provide higher performance. Industrial requirements explicitly state that older systems must
coexist with more modern ones, which pose difficulties when requiring substantial performance improvements. We have formulated the following questions to address these problems:

Q1 Is it possible to increase the overall message processing performance by using message compression?

Manual configuration is frowned upon for large-scale systems with changing message content. Industrial systems implicitly require auto-configurable messaging systems, removing the need for off-line decisions. Communication mechanisms must be able to handle different initial scenarios and a dynamic environment.

Q2 Can an automatic method provide the best message processing performance by selecting the best compression algorithm from a set of algorithms?

Q3 Can an automatic method continuously ensure the best message processing performance when message content changes?

Message compression implies that we are trading CPU resources for a reduction in message size, which in turn leads to quicker message transit time. A message processing node in our industrial environment runs several essential services that must not have their execution disrupted. Since unrestricted message compression may overload the CPU, there must be an overload protection for message compression. We formulate this in the last question:

Q4 Is it possible to limit CPU resources, used for message compression, so that it does not seriously affect other services co-located on the same CPU?

In Section 2 we give an overview of the general ideas presented in this paper. We describe the typical use cases where automatic message compression can be applied and the practical results of the current implementation. Our main contributions are:

- A novel automatic mechanism that automatically and transparently evaluate the messaging performance of different compression algorithms and select the most efficient one for the current communication stream. (Section 3.1)
- Our automatic selection mechanism detects network congestion and message content changes and will continuously select the best compression algorithm. (Section 3.2 and 3.3)
- Our selection mechanism can simultaneously handle multiple communication streams. Each stream will have its own environment providing the possibility to have different compression algorithms for different communication streams depending on their suitability. (Section 3.4)

We have addressed the overload problem by dynamically adjusting the available CPU resources allocated for compression:

- We have implemented a Proportional Integrative Derivative (PID) feedback controller that evaluate the CPU usage and dynamically adapt the ratio of compressed and uncompressed messages. (Section 3.5)
We have tested our implementation on a large telecommunication system with a major market share. We have used a realistic test setup to show that our implementation adds performance improvements to an already existing system.

- We have used production data in the test environment. (Section 4.1)
- We have evaluated 11 different compression algorithms and configurations with varying characteristics. (Section 4.2)

In Section 5 we display the test results from an experiment using data from a telecommunication production system. The first results, in Section 5.1, shows a 9.6% decrease of message Round Trip Time (RTT) when using the automatic compression algorithm selection method compared to not using compression. In Section 5.2 we show that it is possible to select automatically the compression algorithm providing the lowest message RTT. We have extended the automatic selection technique in Section 5.3 showing a message-stream change during the test case execution. The content change is detected, and a different compression algorithm is selected to keep the lowest RTT. We have also shown that the selection process can handle CPU overload situations, see Section 5.4. During the overload situation, message compression is temporarily and proportionally reduced. As the CPU load returns to a lower non-overload level, message compression is resumed to the initial level. We have also reviewed how our results compare to related work, see Section 5. Previous publications use either very few compression algorithms \cite{5} or specialize on explicit types of communication patterns \cite{6}, for example, patterns such as data properties, message length, etc. Such static configuration contrasts with our automatic selection mechanism that automatically adapts to the execution environment without any manual intervention. We conclude the paper with future work, Section 7, and conclusions in Section 8.

1.1. Definitions

We define **host** in our communication system to be a computer that first receives a message and then spends some time processing it producing a result to be later sent onwards to another host. A host is usually, for cost-savings reasons, also configured to handle additional concurrent tasks, such as statistics measurements, user interaction, database management and other similar actions. Further, cost-effective considerations encourage a move of software previously implemented on separate hardware to shared processing boards. Such software consolidations put high demands on the CPU usage of applications sharing a resource, which further complicates the execution environment.

We define **messaging performance** as a function of the time between the \texttt{snd_msg()} call until the receiver obtains the message, i.e. $t_s + t_{t1} + t_r$ in Figure 1a. The messaging performance varies depending on properties such as link speed, host distance in the network. In the case of message compression, additional properties can be added such as compression/decompression rate and compression ratio.
The concept *message processing performance* is the measurable ability to process messages per time interval. In our study, we measure this as the message RTT between two interconnected nodes.

We define the *best compression algorithm* to be the one that gives the lowest message RTT.

2. Problem Formulation and System Model

The main idea presented in this paper is to reduce message round trip time by using selective message compression. Manually finding the best compression algorithm is both complex and time-consuming. There are also practical considerations such as how to handle the scenario when the message-stream content changes. Our approach is to implement a selection mechanism that automatically chooses the best compression algorithm, depending on current message content and network congestion levels. For systems with no compression, this is a straight-forward procedure depicted in Figure 1a.

Process A in Figure 1a communicates with process B, located on another processor. A and B use legacy functionality without message compression. Using message compression makes it possible to increase messaging performance, see Figure 1b.

With our novel solution, we can transparently improve messaging performance by adding message compression to the legacy Application Programming Interface (API). We suggest to add selective message compression functionality inside the `snd_msg()` function and corresponding decompression functionality in `rcv_msg()`. It is a substantial cost-saving for any industrial software to improve performance without API changes. API changes are usually frowned upon as they usually have a significant impact on other software using the API. Breaking a legacy API will often require costly modifications to other system components as well as complex regression testing and challenging customer discussions. Our solution is to update transparently the existing API to monitor each communication instance. Process $A \rightarrow B$ is one instance and process $A \rightarrow C$ is another,
where \( A, B \) and \( C \) are processes in the communication system. The communication API can transparently utilize the most suitable compression algorithm for different instances and types of message content.

The message scenario starts in a round-robin fashion by sending messages using all compression methods. Each communication instance stores compression and network statistics. After an initial evaluation period, one compression algorithm will be selected if it is predicted to give the lowest message round trip time.

The total time \( t_{t1} \) in Equation 1 is the time inside the send function \( (t_s) \), to travel on the link \( (t_l) \) and inside the receive function \( (t_r) \). See Figure 1a.

\[
t_{t1} = t_s + t_l + t_r
\]  
(1)

As illustrated in Figure 1b our approach is to add compression \( (t_c) \) and decompression \( (t_d) \), see Equation 2.

\[
t_{t2} = t_c + t_s + t_l + t_r + t_d
\]  
(2)

The send and receive function does not differ between Figure 1a and 1b, therefore we assume that \( t_s = t_s \) and \( t_r = t_r \). This means that message compression is beneficial if the time it takes to compress and decompress messages is lower than saved link time:

\[
t_c + t_l + t_d < t_{t1}
\]  
(3)

We can predict the performance of each compression algorithm by using statistics gathered from prior communication. The statistical data is then used to predict future message compression, send, link, receive and decompression time for each compression algorithm. The prediction method selects the compression algorithm that gives the lowest message time \( (t_{t1}) \). The selected algorithm is used for a majority of messages until a different algorithm outperforms the current one. To make sure that it is possible to detect a network- or message content change some messages are sent using other compression algorithms. The idea is to gather statistical data for all implemented algorithms, not only the one that is selected.

We assume that \( F \) is the selected compression algorithm for a message stream. \( F \) is, therefore, used for the majority of the transmitted messages. The rest of the algorithms will continuously be evaluated by compressing a small number of messages. If the content of the message stream changes, the algorithm evaluation may favour a different algorithm, which will then be chosen as the best one.

If the CPU load increases beyond a limit, a PID-feedback controller reduces the message compression time-quota, causing messages to be sent uncompressed. Consider the following scenario:
We assume that process $S$ shares the same CPU as process $A$. The throttling mechanism will reduce the compression quota if the load of $A$ and $S$ together exceeds a predefined limit. For example, reduce the number of compressed messages when $S$ performs time-consuming calculations and $A$ solicits heavy compression. Such overload handling mechanism aims to increase the availability of service $S$.

We think that it is hard to manually, in an offline manner, consider all possible scenarios and selecting the most suitable compression algorithm. Our approach, using an automatic mechanism, greatly simplifies this task and provides the flexibility needed for a changing message stream while at the same time being able to provide CPU resources for other services sharing the same hardware resources.

3. Adaption

In Section 2 we demonstrated that it is beneficial to use message compression for particular situations, see Equations 1–3. Up to now we have assumed that the optimal compression algorithm is known. However, in many scenarios, this is not the case. The message content may be unknown to the programmer, which makes it difficult to select an appropriate compression algorithm manually. To find the most suitable compression algorithm there are two approaches. The first method is to manually choose the compression algorithm that the operator thinks is the best one. The second method, the one we suggest in this paper, is to use an automatic selection mechanism to evaluate automatically and select the compression algorithm that is most suitable for the current message stream.

3.1. The Communication Procedure

We explain our automatic message compression method by showing the send and receive procedure. All messages being sent belong to a message stream, which is owned by the sending process, see Figure 2. The message stream is divided into rounds separated by an assessment period where the previous round is evaluated. The system decides the compression strategy for the next round at the end of each evaluation period. It is desirable to keep the round as long as possible while still letting adjustments take effect in a reasonable time. Additionally, it must be long enough so that there is sufficient information to evaluate to make a good estimation of the compression algorithm performance.
Sending a message

The list below summarizes our suggested communication procedure. It shows the major steps and links to later sections describing more details. For each new round $r$ do:

1. Evaluate previous statistics and assign $r$ its parameters.
   
   (a) Calculate the algorithm distribution for $r$ by setting a weighted probability, see Section 3.4. A more efficient algorithm gets a higher probability for being selected than other algorithms.
   
   (b) Derive a compression time budget for $r$, see Section 3.5.

2. Send messages during round $r$.
   
   (a) Select a random compression algorithm according to its weighted probability. This means that there might be different compression algorithms for adjacent messages depending on message size and content. If the time budget is empty, our mechanism will not compress the message.
   
   (b) Send the message.
   
   (c) Update the statistics for the compression algorithm used and decrease the available time budget reflecting the time it took to compress the current message.

3. Until end of round, goto item 2

4. When the round ends, goto item 1.

The round-length is determined by any quantifiable metric, for example, wall-time or number of messages. Which metric to choose depends on system requirements and communication system behavior.

Receiving a message

The receive-procedure is simpler than the sending procedure since it does not require any compression algorithm selection. The following procedure outlines the necessary steps to be performed by the receiver.

1. The message is received.

2. The header of the received message is parsed to reveal the compression algorithm used by the sender.

3. The message is decompressed.
   
   (a) Memory is allocated for the uncompressed message.
   
   (b) The time it took to decompress the messages is stored in a database. The stored time will be used for probe message handling, Section 4.4.

4. The application can resume operation with the newly received message.

The communication procedure is designed to be completely transparent to the application. The legacy compliant API hides all functionality related to message decompression.
3.2. Network Measurements

Our implementation measures the network capacity by continuously monitoring the send-time, $t_s$, and Round Trip Time (RTT), $t_{rtt}$. The send-time is measured inside the `sndmssg()` API function call. Periodic probe messages are sent to measure the RTT. Measurement data is stored in a local database and used in the compression algorithm selection process. Each measurement is a cumulative moving average to ensure that network changes, such as congestion, will influence the algorithm selection procedure. In general, hard compression is favored on slower, congested networks with high transmission time. Using cumulative average data has proven to be successful for our target system. If another system has different demands, it is easy to change the statistical model. Only looking at recent messages may give quicker response time at the expense of not being as resilient.

3.3. Compression Measurements

Several algorithm specific metrics are measured such as compression rate, $t_{cr}$, decompression rate, $t_{dr}$, compression ratio, $r_c$. The counters are updated each round and shows messaging properties for the system it is running on. The values will differ depending on the hardware it runs on and the message content. The values are calculated as cumulative moving average to provide a good balance between quick response and stable behavior. If a particular system experiences problems with this approach is is simple to change this mechanism to a more appropriate one.

3.4. Selecting the Best Compression Algorithm

It is possible to use many different approaches when selecting the most appropriate compression algorithm. For example, we could make a static selection based on the result of an initial evaluation run. Such predefined selection would initially provide the best possible overall compression ratio but any content change in the communication stream would over time cause the selected algorithm to perform poorly. At the other end of the spectrum, we could use all available algorithms evenly in a round robin fashion. Using round-robin provides a lower overall compression ratio but will be robust whenever there are content changes. For comparison, we have defined three different algorithm distributions: Majority, One Algorithm or Round Robin. They are explained in more detail below.

**Compression Selection 1 - Majority Distribution**

We start by identifying the best compression algorithm providing the shortest message RTT. The majority distribution uses the best compression algorithm for as many messages as possible while still retaining the ability to switch to another compression algorithm should the message stream change. At the beginning of each round, a compression algorithm distribution is calculated based on the measurements during previous rounds. The distribution is a suitability rating of the compression algorithms and takes many parameters into consideration,
such as compression time, transmission time, etc. Message stream content can change depending on the users of the communication channel. A change in message content may lead to vast differences in compression ratio and compression time for different algorithms. This is one reason for allowing all compression techniques to run in each round. Completely turning off a compression algorithm would make the strategy static and unable to cope with a changing situation.

We use a simple scheme to decide the compression distribution where the bulk of messages are compressed using the best algorithm and all other algorithms each receive 1% of the compression budget. Such distribution allocation causes the majority of compression time quota to be allocated for the algorithm causing the shortest message round-trip time. We have selected this distribution because it is simple while still being reasonable efficient. Should the need arise, it is always possible to implement more advanced and tailored distributions.

**Compression Selection 2 - One Algorithm**

One algorithm is manually selected and gets the complete compression budget. No other compression algorithm will be used thus making this an offline method. If it is possible to find the best algorithm, it will provide the best compression ratio. However, it is a static selection; which may result in poor performance for message streams where the content changes.

**Compression Selection 3 - Round Robin**

Apply an equal share of compression budget between the available compression algorithms. Each algorithm gets the same amount of slots with no regard to their individual performance. Using all algorithms equally results in great flexibility, but lower performance since it is not fully utilizing the best compression algorithm.

### 3.5. Compression Throttling

We want to make sure that all processes get a fair chance of running. In some cases, high CPU-load and excessive message compression may starve other services running on the same CPU. We have implemented a control algorithm that throttles the amount of CPU cycles that can be used for message compression. The idea is to continuously monitor CPU-usage and current communication bandwidth. If the current CPU-load is low and the desired bandwidth is higher than what is available, it is possible to trade computational capacity for an increased compression level causing the perceived bandwidth to increase.

**Compression Time**

We can measure the time spent compressing messages during a round. This is achieved by looking at the compression time $t_{c(n)}$ and decompression time $t_{d(n)}$ time for each individual message $n$. Adding $t_{c(n)}$ and $t_{d(n)}$ for all $N$ messages during a round results in the total time, $t_{tot}$, spent performing compression activities (4). Our mechanism measures the decompression time and piggybacks
the result to probe messages that are sent between the sending- and receiving node.

\[ t_{tot} = \sum_{n=0}^{N-1} (t_{c(n)} + t_{d(n)}) \]  

(4)

**Controlling the total compression time**

We want to throttle the total time spent compressing messages, \( t_{tot} \), per round to adjust the computational capacity spent for message compression. We use a feedback control algorithm that maximize the amount of time assigned for compression to match the desired CPU-load target level. We want to find the optimal distribution of CPU-load between computational capacity and messaging compression to maximize the throughput without overloading the CPU.

**The Control Algorithm**

We use a Proportional Integrative Derivative (PID) controller to restrict the CPU capacity assigned to message compression. We have implemented this functionality because message compression can easily overload the CPU if used for all messages. An overload situation can seriously affect the functionality of other processes executing on the same CPU. A system engineer can define the amount of CPU capacity assigned to message compression depending on the desired system requirements.

The PID control algorithm uses CPU load as input and \( t_{tot} \) as output. The control algorithm will continuously adapt \( t_{tot} \) to converge to the target CPU load given at the time of system configuration. Increasing \( t_{tot} \) will cause more CPU time to be allocated for message compression.

**Initial Values**

The initial setting for the compression budget (\( t_{tot} \)) is set to 0 since we do not have any knowledge of the message stream content when a new system is deployed. Initially, for the first round, there will be no compression and for the subsequent rounds the feedback control loop will increase \( t_{tot} \) to gradually allow more compression. The desired CPU load is system dependent and, therefore, configurable. Most systems will have different services running at the same time as the messaging system so the CPU-load value must be carefully determined.

**4. Test System Setup**

We have tested our adaptive message compression implementation by running several experiments on a test setup designed for large-scale telecommunication systems. In the following section we will map the generic description in Sections 2 and 3 towards our implementation. The implementation is individual for each target system since the procedure to obtain metrics may differ depending on hardware and operating system.
Figure 3: The structure and behavior of two interconnected nodes in a communication system. Process A and B have multiple message streams using different message compression algorithms depending on their payload content.

4.1. The Test System

We have followed the guidelines and checklist suggested by Petersen [7] to explain the system we have investigated. We have investigated a telecommunication system [8, 9] where the platform consists of a third party embedded real-time operating system (OS). On top the OS approximately 5M Source Lines Of Code (SLOC) implements the middleware. The middleware implements cluster control, error management, upgrade functionality and other generic tasks that are common for all types of applications running on top the OS. Our test system consists of multiple CPU boards where each CPU runs several thousands of processes, see Figure 3. Each process can have its own message queue which stores metadata such as compression algorithm statistics and network congestion levels. The statistical data is used to predict the optimal compression strategy. The result is that two message streams may use different compression algorithms depending on their message content, destination node and network congestion level. Thousands of engineers developed the system [4] for many years. Multiple stakeholders compete for the inclusion of their functionality within the platform. As a result, from the size and complexity, there are many months from the requirement phase to the final release of the implemented functionality. The platform runs on more than 20 types of hardware boards with varying complexity and performance levels, ranging from single-core CPUs with MBs of memory up to large multi-core clusters with many GBs of memory. They have different functionality and hardware layout servicing both voice and data communication. The telecommunication system we have investigated has about 25% market share for telecommunication network equipment and ranks as being the most used radio access network in the world [10]. For our experiments, we are using production control traffic gathered from a production environment intercepted by Wireshark [11]. We replay these messages to provide a real-life scenario. We have implemented the functionality described in this paper as part of a Linux-based system. The application we are targeting is running inside an
emulation layer that adds support for a legacy operating system. We have only made changes in the communication-API by adding message compression support before entering the TCP/IP layer inside the Linux kernel. We have made no changes at all to the Linux kernel itself. The test setup is as follows: Node A is a Quad-Core AMD Opteron 2378@2.4GHz running Linux 3.2; Node B is an Intel Core i7-4600U@2.10GHz running Linux 3.2. A 100Mbit Ethernet network shared with a large number of other workstations connects both nodes.

4.2. Compression Algorithms

There are numerous compression algorithms, all designed for various uses and with different characteristics. Some [12][13] of the algorithms focus on pure compression ratio with little regard to the compression- and decompression rate. Others [14][15] provide a lower compression ratio but are faster. It is easy to understand that both approaches are useful in different situations. Additionally, it is possible to accelerate partial or complete compression algorithms by implementing them in hardware [16]. In this paper we focus on lossless compression thus completely disregarding lossy compression techniques. Lossless techniques could be applied to data that doesn’t need to be transmitted in an unchanged manner. Our implementation of the automatic mechanism uses eleven compression algorithms/configurations; LZFX[17], LZO[14], LZO-SAFE which is a safe configuration of LZO, LZMA[12], LZW[18], BZ2[13], LZ4[15], FastLZ level 1 and 2 [19], Snappy[20], and QLZ[21]. The key properties are listed in Table 1. The list is extended from the results by Karlsson and Hansson [22] comparing several compression techniques with regard to compression ratio and resource usage. Their work also investigates the suitability for each algorithm in the context of communication scenarios. Ringwelski et al. [23] manually investigates a number of compression techniques with regard to compression ratio and computational resources. This is the starting point for our investigation, how can this task be fully automated?

To measure the performance of each compression algorithm we define three key properties: compression time (Definition 1), decompression time (Definition 2) and compression ratio (Definition 3). A compression ratio $r_c \leq 1$ indicates that no compression is achieved or even worse that the resulting message is larger than the uncompressed. Achieving a higher compression ratio $r_c > 1$ results in a smaller compressed message compared to the original one.

**Definition 1.** The compression time, $t_c = s/t_{cr}$ is defined as the time to compress a particular message of size $s$. The compression rate, $t_{cr}$, is the speed of compression, [B/s].

**Definition 2.** The decompression time, $t_d = s/t_{dr}$ is defined as the time to decompress a particular message of size $s$. The decompression rate, $t_{dr}$, is the speed of decompression, [B/s].

**Definition 3.** The compression ratio of a particular algorithm is defined as $r_c = s_u/s_c$, where $s_u$ is the size of the uncompressed message and $s_c$ is the corresponding size when being compressed.
Table 1: Implemented compression algorithms and their characteristics.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>LZFX</td>
<td>Fast compression, low (c_r)</td>
<td>[17]</td>
</tr>
<tr>
<td>LZO</td>
<td>Fast compression, low (c_r)</td>
<td>[14]</td>
</tr>
<tr>
<td>LZO-SAFE</td>
<td>A safe, slightly slower, configuration of LZO.</td>
<td>[14]</td>
</tr>
<tr>
<td>LZMA</td>
<td>Slow compression, high (c_r)</td>
<td>[12]</td>
</tr>
<tr>
<td>LZW</td>
<td>Medium compression, medium (c_r)</td>
<td>[18]</td>
</tr>
<tr>
<td>BZ2</td>
<td>Medium compression, high (c_r)</td>
<td>[13]</td>
</tr>
<tr>
<td>LZ4</td>
<td>Fast compression, low (c_r)</td>
<td>[15]</td>
</tr>
<tr>
<td>FastLZ lv1</td>
<td>Fast compression, low (c_r), suitable for small messages.</td>
<td>[19]</td>
</tr>
<tr>
<td>FastLZ lv2</td>
<td>Slightly slower than lv1, higher (c_r) than lv1.</td>
<td>[19]</td>
</tr>
<tr>
<td>Snappy</td>
<td>Very fast compression, medium (c_r), suitable for text messages.</td>
<td>[20]</td>
</tr>
<tr>
<td>QLZ</td>
<td>Very fast compression, medium (c_r), suitable for small messages.</td>
<td>[21]</td>
</tr>
</tbody>
</table>

Adding new compression algorithms is simple. The current implementation uses a list where the compression algorithms are transparently used. All algorithm measurements are generic and they do not depend on the specific implementation.

4.3. Putting it All Together

The message stream is divided into communication rounds, see Figure 2. As described in Definition 4, each round consists of two phases, evaluation and transmission. Transmitting more messages each round reduce the relative cost of the evaluation phase but also the ability to detect a message content change. In our target system we have empirically defined a round to 1000 messages.

**Definition 4.** A communication round is started by an evaluation phase followed by a transmission phase, and it is delimited by a fixed number of messages.

The first part in a round is the evaluation of statistical data retrieved from previous messaging rounds, see Sections 3.2-3.3. The evaluation procedure uses historical data to predict if message compression should be used for subsequent messages, as defined by Equation 2. The most suitable compression algorithm is chosen depending on its ability to reduce the message transition time. During the second part, see Figure 2, messages will be compressed using the algorithm distribution decided by the selection phase.

**Estimating the Compression Metrics**

The time to transfer a message between two interconnected nodes is a central concept in this paper and is denoted transmission time, see Definition 2. We have split this procedure into several parts that are individually measurable in a run-time environment. The first part is the sending time, see Definition 5. It
is the time it takes to prepare the message and present it to the driver that will perform the actual link transmission.

**Definition 5.** Sending time is the time spent inside the `snd_msg()` function call. The time to send a message is defined as, $t_s = s / t_{sr}$, where $s$ is the size of the message in Bytes and $t_{sr}$ is the send rate [B/s].

The second part is the Round Trip Time (RTT), see Definition 6.

**Definition 6.** The round-trip-time (RTT), $t_{rtt} = s / t_{rtr}$, is defined as the time it takes for a message of size $s$ to travel from node A to node B and back to A. The amount of data per time unit sent back and forth between two communication partners, RTT rate, is defined by $t_{rtr}$ [B/s].

**Estimating the Transmission Time**

With Equations 1–3 in Section 2 we can describe the time it takes between the sending application calls the `snd_msg()` function and the receiver obtains the decompressed message and can operate on it. For our system we assume that the link time ($t_l$) is roughly half the RTT, as defined by Equation 5. We assume that it takes equal time back and forth between two nodes, which gives an estimation for our proposed algorithm.

$$t_l = \frac{t_{rtt}}{2}$$  \hspace{1cm} (5)

Combining Equation 2 with Equation 5 results in Equation 6.

$$t_t = t_c + t_s + \frac{t_{rtt}}{2} + t_d$$  \hspace{1cm} (6)

**Initial Configuration**

The compression algorithm evaluation procedure needs statistical data to be able to predict the need for message compression and calculate the distribution over available compression algorithms. No predefined statistical data is present at the start of the first communication round (in the system lifetime). By using all compression algorithms in a Round-Robin (R-R) manner, initial data is obtained. We have empirically chosen 10 seconds for our target system since it gives ample measurements to use in the first iteration of the selection process. Initial algorithm selection jitter is also filtered out by running R-R during the first part of communication. In our target-system the message content changes but not very frequent, which makes it possible to use the first set of messages as an indication of near future content of the message stream. Later message stream changes will naturally affect the compression algorithm selection if needed. Each algorithm will have its own set of measurements for compression time ($t_c$), send time ($t_s$) and decompression time ($t_d$). The round trip time ($t_{rtt}$) is a network-dependent parameter stored on a per-network basis.
4.4. Real-World Compression Throttling

Compression throttling, see Section 3.5, is dependent on the ability of accurately measuring the current CPU usage of the system. We assume that people interested in utilizing this algorithm have such metrics available. The CPU load is used as input to the automatic throttling functionality when determining actions for current and future messaging strategies. In particular, we are interested in the current CPU-load and communication properties since these are used to determine the compression technique. If the CPU is already saturated with other computational tasks we will avoid burdening it further with compression to allow the highest possible throughput while preserving the availability of other services.

Measuring the CPU-load

The CPU load is measured on a per-core basis, see Definition 7, and is used by the feedback control loop determining how much time should be spent on compression. Normally the target CPU load should be set to a value that allows other services to run in the desired way. Setting the target CPU load too high may cause the system to overload since all messages will be compressed and the worst case is that some process may starve out vital functionality on the system.

Definition 7. The CPU-load, $L$, is defined as the number of processes, ready to execute, in the run-queue of the operating system.

We use the 1 minute running average for the CPU-load to avoid jitter. The load values are retrieved from the Linux system by reading /proc/stat. This greatly reduces oscillation problems in the feedback controller, where intermittent background service usage may cause spikes in the CPU-load. To measure other parameters such as round-trip-time, RTT, we issue periodical probe messages to other nodes. These messages are used to determine how saturated the network currently is.

5. Results

We have answered the research questions, see Section 1, by performing experiments on a test system that emulates a large telecommunication system. As experiment data, we are using intercepted control traffic from a production environment, gathered by Wireshark [11]. The production node was fully operational handling control plane traffic and other routine maintenance tasks while we were capturing the experimental data. The test data excludes data plane traffic such as call or video data. Our assumption is that we will provide a more realistic evaluation of our suggested techniques when using production traffic data compared to synthetic test data. Even though we have tested our implementation on a particular type of system, the proposed mechanism is adaptable to any other kind of system. Each test has followed the process defined in Section 3.1, including the initial 10-second round-robin execution of all algorithms.
Q1 Is it possible to increase the overall message processing performance by using message compression?

In this experiment, we show that our online mechanism automatically finds the most efficient compression algorithm from a set of available algorithms, depending on the content of the message stream. We have run three different test setups within this experiment. All tests send an equal amount of data but with different content and different settings for the automatic compression algorithm selector.

The first test is a reference measurement using production data, see Figure 4a. We have temporarily suspended message compression by disabling the selection mechanism. This scenario shows the original production system behavior when not using any message compression.

In the second test, we use a synthetic data set containing only zeros, see Figure 4b, together with the automatic compression selection mechanism enabled.
Snappy [20] is a fast and efficient compression algorithm for simpler textual messages. It is selected as the best compression algorithm for this synthetic data set because it outperforms all other algorithms. Network disturbances at 7k–10k messages in Figure 4b triggers an additional compression algorithm selection. When the network has returned to a stable state, Snappy is once more selected and finally results in a 14.0% reduction in RTT compared to the first test. This experiment shows that different compression techniques can be chosen depending on the transmitted message data.

The third and final test, shown in Figure 4c, uses production data together with the automatic compression selection mechanism. In this experiment, the selection mechanism chooses QLZ [21] as the most efficient compression algorithm with an 8.9% reduction in message RTT over uncompressed messaging. We have decided to connect the test nodes to a busy network to provide a more dynamic environment. External network congestion causes the RTT transient at 20k messages.

5.2. Algorithm Selection Methods

Q2 Can an automatic method provide the best message processing performance by selecting the best compression algorithm from a set of algorithms?

As described in Section 5.1, the automatic selection mechanism selects the best compression algorithm depending on the message content and the network environment. We have designed a test to demonstrate the algorithm selection methods by replaying data from a production node. Figure 5 shows how RTT converges towards a specific value over multiple communication rounds. QLZ [21] is automatically selected as the most efficient compression algorithm for this test setup because it provides the best balance between compression ratio and compression rate. The automatic mechanism is inferior to uncompressed messaging.
Table 2: Relative improvements for different algorithm selection strategies.

<table>
<thead>
<tr>
<th>Selection Mechanism</th>
<th>RTT [ms]</th>
<th>Relative Time Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uncompressed (Reference)</td>
<td>1.57</td>
<td>0.0</td>
</tr>
<tr>
<td>Round Robin</td>
<td>1.45</td>
<td>-7.6%</td>
</tr>
<tr>
<td>Automatic</td>
<td>1.42</td>
<td>-9.6%</td>
</tr>
<tr>
<td>Manually selecting QLZ as the best</td>
<td>1.35</td>
<td>-14.0%</td>
</tr>
<tr>
<td>compression algorithm during the first part</td>
<td></td>
<td></td>
</tr>
<tr>
<td>of the test, between 0–6k messages. At 6k</td>
<td></td>
<td></td>
</tr>
<tr>
<td>messages, the QLZ performance surpasses</td>
<td></td>
<td></td>
</tr>
<tr>
<td>uncompressed messaging resulting in a 9.6%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RTT reduction after a total of 50k messages.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

As a comparison, a manual offline selection of QLZ as the compression algorithm results in a 14.0% reduction in RTT, see Table 2. The current cost of automating compression algorithm selection (14.0% – 9.6% = 4.4%), which can be attributed to intermittent use of non-optimal compression algorithms to be able to detect changes in the message stream. It is possible to reduce the selection cost by increasing the usage of the best compression algorithm while reducing the usage of other algorithms. Changing the ratio between the selected and other algorithms may reduce the ability to detect changes in the message stream. In our setup, we have empirically decided to assign 1% of the compression quota to the non-optimal algorithms and the remaining quota to the best algorithm. The ratio between the best algorithm and the rest is easily configurable and dependent on the desired system behavior.

5.3. Automatic Algorithm Selection for Changing Message Streams

Q3: Can an automatic method continuously ensure the best message processing performance when message content changes?

In this experiment, we show that our automatic mechanism can select different compression algorithms when a message stream changes. The mechanism initially selects Snappy [20] for text messages and later selects QLZ [21] for production node data. Figure 6 shows the measurements when changing the message stream content after 10k messages. From 0–10k messages, the data set contains a zero pattern. At 10k messages, the data set is switched to production node data. The first and uppermost graph, Figure 6a, shows the RTT, which at 5k messages is lower than uncompressed messaging. The second graph, Figure 6b, shows the cumulative average compression ratio for all evaluated compression algorithms. Snappy is chosen for the first data set and provides a low RTT for the system. At 10k messages, the message stream switches to production data the QLZ compression algorithm performs better and is subsequently selected at 15k messages. The third graph, Figure 6c, shows the suitability of each compression algorithm. The fourth graph, Figure 6d, shows the cumulative number of messages compressed by each algorithm. Before 15k messages,
Figure 6: Message stream change at 10k messages triggers an re-selection (Snappy→QLZ).
the Snappy algorithm is used for most messages. After 15k messages, there is a selection time where the best algorithm is switching between several ones. From 30k messages, the QLZ algorithm is used as the sole compression algorithm. The final graph, Figure 6e, illustrates the relative compression time for each algorithm.

5.4. Overload Handling

Q4 Is it possible to limit CPU resources used for message compression so that it does not seriously affect other services co-located on the same CPU?

We have, in this experiment, shown that our feedback control algorithm can handle overload situations. We have divided the overload experiment result into three intervals, shown in Figure 7. In the initial interval, between 0 and 35k messages, QLZ is selected as the most appropriate compression algorithm.

The second interval, starting at 35k messages, shows a CPU overload situation triggered by using manually starting Cpuburn [24]. Cpuburn is designed to evaluate system load by creating massive CPU load. Due to the compression time-quota reduction during the second interval, our mechanism reduces the number of compressed messages. The feedback control algorithm detects the overload situation and reduces the time quota assigned for compression, which frees CPU-resources for other applications executing on the shared resource.

The third interval starts at 50k messages after terminating the Cpuburn application, and the overload situation ends. The QLZ compression algorithm usage increases at 60k messages after gradually restoring the initial CPU load level.
6. Related Work

We have split the related work into three subsections. The first details message compression from a general perspective. The second section introduces different compression algorithm selection methods. More specifically, how to find the algorithm that provides the best messaging performance. In the third and last part, we summarize the state-of-the art related to overload handling.

Message Compression

Several earlier publications and implementations propose that compression can improve the overall communication performance. Wiseman et al. [25] investigate loss-less compression of communication systems. They benchmark for each compression algorithm using off-line data and use these measurements to automate algorithm selection. Gutwin et al. [6] describe a transparent way of compressing Groupware messages in an efficient way. Their method is convenient and easy to use for framework users since it supports both text and serialized objects.

Nicolae [26] apply compression to cloud computing and investigates its effect on cloud storage. The Grid5000 research network has tested the implementation and report a significant network traffic reduction when using LZO and BZIP2. The trend in recent CPUs is to include hardware support for compression and decompression [16]. Hardware support offloads the CPU and can improve the overall system messaging performance. The selection mechanism presented in this paper will treat a hardware supported compression algorithm the same way as a software algorithm.

The compression techniques described above adopts a semi-passive behavior by selecting the compression algorithm during configuration time. If we have prior knowledge of the system behavior, a static selection of compression algorithms can be acceptable or even desirable since it provides determinism to the system. For other types of systems, such as our dynamic telecommunication system, the communication streams change over time, and their content is difficult to predict in advance. Providing a way to support a dynamic message content is one of the major reasons for implementing our online compression algorithm selection methods.

Compression Algorithm Selection Methods

Jeannot, Knutsson, and Björkman has created a suite of papers [5, 27, 28] that describes an implementation using adaptive message compression. They have changed the Linux communication mechanism to compress/decompress messages as a function of available memory and communication capacity. Their implementation uses predefined compression levels compared to our approach that continuously evaluate all algorithms. Jeannot, Knutsson and Björkman [28] re-implements previous techniques in user-space to improve portability. This method is similar to ours where a user-space API hides message compression. In a later paper Jeannot [29] describes an official library that supports automatic message compression. The AdOC-library is freely available at the official
web page [30]. There are some differences compared to our work: The first is that AdOC uses POSIX standard calls while we have adapted a legacy compliant asynchronous messaging system. The AdOC implementation uses multiple threads to compress and communicates data. Our implementation executes the compression in single threaded user-mode to isolate the functionality to a single execution context. Isolation is essential to reduce inter-thread communication in the current system implementation. Furthermore, AdOC uses large (200kB) buffers compared to ours that are usually multiple of 1000s of bytes. Our assumption is that coalescing multiple messages in larger chunks would increase the message round trip time. Our target system would suffer a performance impact if the message round trip time increased. Jeannot’s implementation monitors the send queue length. The network is saturated if the queue grows, which means that the message stream needs higher compression ratio. Adoc pre-defines a set of compression algorithms, for which each one defines the wanted compression level. Depending on the desired compression level, AdOC selects one algorithm out of the set of available ones. Our method is on the other hand continuously assessing all compression algorithms making it more flexible. In our solution, there is no need for offline algorithm evaluation to determine their suitability with regards to different message streams.

Sucu and Krintz [31] have created a communication environment called Adaptive Compression Environment (ACE). It aims to change the behavior of socket communication by compressing particular type of messages. ACE will only compress messages larger than 32kB. This approach differs to ours where all messages are transparently evaluated to detect if compression can improve messaging performance. ACE uses one compression algorithm, Zlib, compared to our method where we have implemented eleven algorithms. In their later paper, Krintz and Sucu [32] implements additional compression algorithms such as Bzip2, Zlib, and LZO. The main difference compared to our implementation is that algorithm selection depends on offline measurements using training files. Each compression algorithm is profiled to measure its performance. The adaptive online algorithm will then utilize algorithm profile information when selecting the most suitable compression algorithm. Offline operations require the system engineer to have prior knowledge of the message stream content. Our implementation does not require any manual evaluation since it continuously evaluates the compression performance.

Pu and Singaravelu [33] describes the trade-off between available bandwidth and the required computational capacity when compressing messages. They present a thorough investigation of simple schemes such as “compress-all messages” or “compress-none”. Gray et al. [34] points out that it is hard to decide when to compress messages or not. They expand earlier work by Pu and Singaravelu [33] by including mixed sets of compressed and uncompressed messaging. We employ this technique in our messaging subsystem by both sending uncompressed messages as well as messages compressed with different algorithms.

Brunet et al. [35] describe a technique to auto-tune compression parameters depending on the application hardware. Hardware profiling is performed once for each platform, and the resulting profile is an indicator of successful network
communication parameters. In our algorithm selection process, we continuously monitor the performance of each compression algorithm compared to the offline evaluation proposed by Brunet et al. [35].

Biederman [36] owns a patent describing a general idea of receiving, compressing and sending messages. The method is similar to ours but static. We have implemented a PID feedback controller to manage the CPU quota allocated for message compression. The feedback controller adjusts the CPU load to ensure that other services can coexist on the same CPU. Biederman uses several predefined compression levels that depend on the message content. Biederman contrasts with our solution that simultaneously evaluates several compression algorithms allowing the best algorithm to dominate.

In general, our approach is more flexible than the publications described above. We have provided a messaging subsystem with the primary goal of being automatically adaptable to a changing message stream and system behavior.

Overload Handling

Our target system needs an overload mechanism because there are many co-located services sharing the same hardware. Message compression is computationally heavy and can easily starve other processes. We have managed this problem by automatically constraining the amount of computational capacity available for message compression. There are other ways to control the CPU quota assigned to message compression. Jeannot, Knutsson and Björkman [5, 27, 28] have implemented AdOC, which uses the send-queue length as an indication of the desired compression level. AdOC deduces that higher compression ratio is required when the send queue gets full. Message compression will take longer time if the processor is busy with higher prioritized tasks, resulting in a small number of messages in the send queue. A low message count in the send-queue triggers a compression-level reduction. Our implementation does not require the same type of low-level kernel modifications needed by AdOC. Avoiding 3PP kernel modifications is a tremendous benefit concerning kernel upgrades and support agreements. Our design is also much more fine-grained since it is possible to specify exactly the maximum CPU-usage. There is a strict CPU usage cap for some industrial systems. Strict overload handling can be beneficial for industrial usage not to exceed such predefined CPU usage limit.

7. Future Work

Automatic message compression can be useful for communication systems by increasing the overall messaging performance. In this paper, we have described one method to provide an automatic and adaptive implementation supporting a changing environment. However, there are many issues that we have not yet investigated. In the following sections, we have described the main topics we would like to investigate further.
System Setup

First, we would like to test our implementation on other types of systems and environments. Expanding our experiments to cover more complex network topologies and other message streams would further affirm our belief that our selection mechanism can work in a general environment. It would also be intriguing to test our implementation running on other types of communication links. Our implementation currently uses a link model that has a long lifetime. How will our selection mechanism handle links that connect and disconnect frequently? Most probably, the empirically defined constants would need to be modified. Changing the round-length and compression algorithm selection distribution can radically alter the behavior of the communication subsystem. We would also like to experiment further with the feedback controller. It is interesting to find controller parameters that reduce the converge time. How will the PID controller handle different environments and other settings for the controller constants?

Compression Techniques

Our implementation is designed to support easy addition of additional compression algorithms. In this paper, we show a set of eleven integrated compression algorithms. These algorithms are sufficient to test our implementation and has provided message performance increase. In some particular circumstances, additional compression algorithms can give better performance. Our suggestion is to implement additional compression algorithms specifically tailored to the use-cases defined by the target communication system. Intel provides, in recent CPUs, hardware support for LZO compression [16]. Our automatic selection mechanism already supports hardware-accelerated compression algorithms. It would be interesting to investigate the performance and effect of mixing software and hardware algorithms. We would also be interested in running the communication system on networks with far greater bandwidth. The performance of our sample implementation suffers from multiple memcpy() when handling messages. It would be rewarding to remove these bottlenecks and adopt a zero-copy approach.

Automatic Compression

We think that the automation of message compression is of great importance. Systems get increasingly more complex and, therefore, difficult to configure. We would like to experiment with other types of compression handling mechanisms. It may, for example, during a round be more efficient to reduce the overall compression level rather than sending some highly compressed messages while leaving the rest uncompressed. We would also like to make use of more metrics for the system utility calculation, such as the receiver CPU-load, multicore CPUs, and several more.

We can coalesce several messages into one larger compressed message, which will reduce much of the overhead related to packet marshalling, memory allocation. Coalescing should provide a significant performance improvement over
the current implementation but would, on the other hand, require a more complicated message receiver.

Little regard is taken to the temporal locality of statistical data in the current implementation. Old message compression statistics weigh the same as recent data, which favors a stable behavior. Recent data should have higher importance to decrease the adaption time when new communication circumstances occurs.

8. Conclusions

We have shown that it is possible to increase the message processing performance of large-scale industrial systems [4] by selectively and automatically compressing messages. We have also shown that message compression can coexist with other services without introducing starvation problems. We have integrated eleven algorithms and compression configurations; LZFX [17], LZO [14], LZO-SAFE, LZMA [12], LZW [18], BZ2 [13], LZ4 [15], FastLZ level 1 and 2 [19], Snappy [20] and QLZ [21]. The system will automatically choose the compression algorithm that provides the lowest message round trip time (RTT). The RTT is affected by factors such as compression rate, compression ratio and link speed. There are also additional external factors that influence the suitability of each algorithm, such as the CPU processing power and the network congestion level. Furthermore, we have implemented a mechanism that continuously evaluates the algorithm suitability. The mechanism detects when the content of a message stream varies by monitoring the algorithm suitability. By continuously evaluate all compression algorithms we provide a robust and fully automatic algorithm selection suitable for large-scale deployment. The automation is particularly suitable for environments where it is hard to manually decide the optimal compression algorithm.

We have also implemented an overload protection to ensure that excessive compression does not starve other services co-located on the same CPU as the communication application. The compression throttling is implemented using a Proportional Integrative Derivative (PID) controller that monitors the CPU usage. When starting the communication application, all messages are sent without compression. The CPU-quota assigned for message compression is increased until reaching the desired max-load. Simultaneously, our communication mechanism continuously evaluates all compression algorithms to find the one producing the shortest message round trip time.

We have implemented the automatic message compression method on a large-scale industrial telecommunication platform [8] with a major market share [9]. We have tested our implementation with production data gathered at customer sites and replayed it in a lab. We show that the automatic compression mechanism yields a 9.6% reduction in RTT when using production data.

Acknowledgment

This work is funded by Ericsson AB and by the Swedish Knowledge Foundation through the ITS-EASY program. Special thanks go to MSc students
Stefan Karlsson and Erik Hansson [22] for implementing and evaluating several of the compression algorithms used in the paper.

References

URL http://cpuburnin.com
[33] C. Pu, L. Singaravelu, Fine-Grain Adaptive Compression in Dynamically

