

Diplomarbeit

Motivations and Challenges in Designing a Distributed Log Management Framework

ausgeführt am

Institut für Softwaretechnik und interaktive Systeme
Information & Software Engineering Group
der Technischen Universität Wien

unter der Anleitung von
Ao.Univ.Prof. Dr.techn. Andreas Rauber
und
Univ.Ass. Dr.techn. Alexander Schatten
als verantwortlich mitwirkenden Assistenten

durch

Robert Fischer
Matr.Nr. 0026899
Würthgasse 12/4, 1190 Wien

Wien, am 16. April 2007

Abstract

Since the beginning of computer technology many methods for logging status, error and debugging information were introduced. The range spans from simple printing of text messages on the terminal to sophisticated logging systems using database backends on distributed servers.

However due to different implementation approaches many of these systems use proprietary protocols for transferring messages between the concerned systems. Especially in heterogeneous systems this incompatibility leads to serious problems when a centralized logging of all system information is desired.

This work introduces the basics of logging and log management and presents guidelines for designing advanced system architectures, capable of managing and processing large amounts of logged data. After analyzing existing logging implementations and discussing their advantages and downsides, the system design of the “XMPP Log Management Framework”, a novel approach for log management will be presented and elaborated by a prototype implementation of a Java based administrative logging console using an extended XMPP instant messaging protocol for message transfer. Based on log4j and extending its highly modular architecture, the framework is able to integrate with legacy systems as well as existing system management environments while allowing standardized and secure configuration and monitoring access through commonly used instant messengers.

Furthermore, possible future trends in the field of log management are outlined by evaluating the benefits of using advanced event processing algorithms in log management systems. Particularly the integration of logging into event-triggered architectures, in order to provide an embedded logging solution for loosely coupled systems as well as the importance of logging in systems administration will be discussed.

Zusammenfassung

Seit den Anfängen der computerunterstützten Datenverarbeitung wurden zahlreiche Technologien zu Protokollierung von Status- und Fehlermeldungen eingeführt. Die Spanne reicht hierbei von der simplen Ausgabe relevanter Meldungen auf die Konsole bis hin zu umfangreichen Logging-Systemen, die auf Datenbanken und verteilten Systemen aufbauen.

Aufgrund unterschiedlicher Ansätze zur Implementierung solcher Systeme wurden meist verschiedene proprietäre Protokolle zur Übertragung von Log-Nachrichten verwendet. Speziell in heterogenen Umgebungen entstehen durch diese Inkompatibilitäten zwischen den Systemen umfangreiche Probleme - vor allem wenn eine zentrale Protokollierung aller anfallenden Meldungen erwünscht ist.

Im Rahmen dieser Diplomarbeit soll eine Einführung in die Thematik des Loggings und in dessen Management gegeben werden, sowie Richtlinien und Beispiele für den Aufbau von Systemen, welche die Verarbeitung und Verwaltung grosser Mengen solcher Daten ermöglichen, diskutiert werden.

Im nächsten Schritt werden die Vor- und Nachteile von bestehenden Log-Management Systemen aufgezeigt und mit dem "XMPP Log Management Framework" ein neuer Ansatz präsentiert, der auf log4j und dem Instant-Messaging-Protokoll XMPP basiert.

Mittels einer prototypischen Teilimplementierung des vorgeschlagenen Systems - einer Java-basierten Administrations-Konsole, welche eine Erweiterung des Instant-Messaging Protokolls XMPP für den Transport von Steuer- und Log-Meldungen verwendet - soll das Konzept evaluiert werden. Durch Einsatz des standardisierten Protokolls XMPP soll aufgezeigt werden, wie durch das vorgestellte Framework und die Möglichkeit, einen beliebigen Instant-Messenger für den sicheren Zugriff auf Logging-Daten und deren Verwaltung zu verwenden, Probleme und Unzulänglichkeiten von existierenden Systemen vermieden werden können.

Die Integration dieses Systems in bestehende Umgebungen wird durch eine Erweiterung der log4j-Plattform, sowie durch die Integration in das JMX-Framework umgesetzt.

Weiters sollen mögliche zukünftige Entwicklungen im Bereich des Loggings, wie zum Beispiel die Kombination mit "event processing"-Algorithmen aufgezeigt werden. Durch die Einbindung von Logging-Systemen in ereignisgesteuerten Umgebungen soll aufgezeigt werden, wie solche lose gekoppelten Systeme von einer integrierten Datenprotokollierung profitieren können. Weiters wird auf die Bedeutung der Datenprotokollierung in der System-Administration hingewiesen.

Acknowledgements

I want to thank Dr. Alexander Schatten for the interesting discussions and his continuous support. I thank Prof. Andreas Rauber for his willingness to advise my thesis.

I would like to thank my parents for giving me the opportunity to study at University. Last but not least, I would like to thank all friends and colleagues for having a good time during my university studies.

Contents

1	Introduction	1
1.1	Outline	1
1.2	What is Logging	1
1.2.1	Log Messages in Detail	2
1.2.2	Transfer of Log Messages	4
1.3	Introduction to Log Management	6
1.3.1	Requirements	7
1.3.2	Log Policy Definition	7
2	Log Management Systems: Current Concepts	13
2.1	Architecture	13
2.1.1	Input Layer	14
2.1.2	Processing Layer	15
2.1.3	Output Layer	16
2.2	Message Processing	18
2.3	Log Data Analysis	21
2.3.1	Introduction	21
2.3.2	Use Cases	21
2.3.3	Analysis Algorithms	23
2.3.4	Log Data Visualization	25
2.4	Summary	27
3	Problems and Conceptual Issues with Current Systems	29
3.1	Interoperability	29
3.1.1	Interoperability at Data Layer	29
3.1.2	Interoperability at Transmission Layer	31
3.1.3	Interoperability at Storage Layer	32
3.2	Dependability	32
3.2.1	Security	33
3.2.2	Reliability	34
3.2.3	Availability	37
3.2.4	Safety	39
3.3	Portability	40
3.4	Scalability Issues	40
3.4.1	Load Scalability	41
3.4.2	Space Scalability	41
3.4.3	Space-Time Scalability	42
3.4.4	Structural Scalability	42
3.5	Architectural Issues	43

3.6	Summary	43
4	The XMPP Log Management Framework	45
4.1	Overview	45
4.1.1	Features	46
4.1.2	Back-Channel Feature	48
4.1.3	Improvements over Existing Systems	48
4.2	System Design	50
4.2.1	System Description	51
4.2.2	Chainsaw Integration	56
4.3	A Real World Example	58
4.4	Summary	61
5	Logging in Event-Driven Environments	63
5.1	Introduction	63
5.2	Log Message Processing in Event-Triggered Environments	64
5.3	Advanced Message Processing and Root Cause Analysis	66
6	Evaluation and Future Work	69
6.1	Advanced Message Correlation	69
6.2	Integration in Systems Management	70
7	Summary and Conclusion	71
	Appendix	73
A	Syslog	73
B	NT Event Log	74
	Bibliography	77

List of Figures

1.1	XML logmessage	3
1.2	Example transfer of a log message	6
2.1	Example log management system	14
2.2	Matching process example	20
2.3	Text outline	26
3.1	Cluster architecture example	38
4.1	An example integration	49
4.2	Log4j overview	52
4.3	The Chainsaw log-viewer	56

List of Figures

1 Introduction

1.1 Outline

This chapter describes the basics of logging in computer systems as well as the structure of log messages and introduces into the concept of log management.

In chapter 2 the detailed architecture of a log management system is explained and the processing and analysis of log messages is outlined.

Chapter 3 reviews problems and conceptual issues a log management system is faced with and provides possible solutions.

In chapter 4 the proposal for a new type of log management system is presented and its architecture and advantages over current systems are described.

Chapter 5 explains how the integration of logging in event-driven environments could provide valuable information for the implemented business processes.

In chapter 6 possible future concepts in the area of log management and system integration are presented.

Chapter 7 provides a summary and draws a conclusion.

1.2 What is Logging

In the world of seafaring the word *log* denotes a device for measuring the rate of a ship's motion through the water. The result of this measurement was registered in a so called log-book, or simply *log*.

Just like in the glory days of seafaring in the field of information technology the word *log* has something to do with recording data or events for later or immediate analysis. In fact, in a technical point of view, the word *logging* denotes the action related to adding data about occurred events or actions to a so called *log*.

These events can be notifications about status changes, error occurrence or for example periodical heartbeat messages, reporting system availability.

It is important however to emphasize that logging is not only used to track down malfunction or for incident investigation, but also to gain status information about a running system or for accounting and statistical purposes.

Adam Sah defines a log as follows:

“Logs are append-only, timestamped records representing some event that has occurred in some computer or network device.” [Ada02]

Although the format or the contained data of a log message adheres to no formal standard, the above definition leads to the fact that a log message should at least contain an attribute containing the source of the message and one containing the timestamp in addition to the logged message data itself. As a matter of fact, nearly all commonly used logging systems use these three essential attributes in their log

messages. In the following chapter the format of commonly used log messages will be explained and the flow of log messages in a logging system will be outlined.

1.2.1 Log Messages in Detail

Although the format of a log message is dependent on the needs of the specific logging system, nearly all log message formats include the following information:

Timestamp The timestamp is an important record in a log message. In addition to the error or informational message itself, the timestamp allows creating a chronological trace of coherent messages. Because log messages typically do not have any specified or inherent correlation to each other, by evaluating the timestamp it is possible to group them chronologically. Such information is essential when debugging a system by only using log messages or trying to reconstruct the course of events causing a failure of a system.

Source Since log messages from multiple systems or applications may be collected in a single repository, it is feasible to include information about their source. Such information might be a hostname, the name of the generating application or any other data unambiguously identifying the source of the log message.

Severity To quickly distinguish important from dispensable messages, almost every log protocol defines a severity field in its log message format. Usually only discrete values like *information*, *error*, etc. . . or numbers are used to indicate the severity. This constraint allows simple categorization of log messages, and is mostly used to easily filter out unwanted messages from a message stream, interpreting the severity as *level*, where levels form a hierarchy, thus allowing for example, to filter out messages below or above a certain level.

Table 1.1 shows some typically used levels

LEVEL	DESCRIPTION
<i>trace</i>	level for lowest priority, typically used to trace the execution flow
<i>info</i> or <i>notice</i>	used for informational events
<i>warn</i>	used to indicate a non critical event
<i>error</i> or <i>alert</i>	used to indicate a critical event
<i>fatal</i> or <i>emergency</i>	used to indicate a fatal event. Typically used when a critical error lead to an unusable system

Table 1.1: Standard log-levels

As mentioned, the levels are hierarchically ordered, so that the following inequation holds:

$$\text{trace} < \text{debug} < \text{info} < \dots < \text{fatal}$$

message Finally, a log message typically includes a record containing either human readable information or binary data related to the logged event itself. This information can be anything from plain text to sophisticated data structures describing for example a stacktrace or other detailed debugging information.

It is important to mention that the logged information is not restricted to tracing or error messages only, but may be virtually any data related to a specific event occurred in the runtime of a process.

To have a better understanding what a log message may look like, figure 1.1 shows an example of a XML formatted log message.

```
<log4j:event logger="org.apache.log4j.xml.XMLLayoutTestCase"
timestamp="Mon Aug 10 21:07:25 CEST 2006"
sequenceNumber="145"
level="DEBUG"
thread="main">
<log4j:message><![CDATA[hi]]></log4j:message>
<log4j:throwable><![CDATA[java.lang.Exception
at org.apache.log4j.xml.XMLLayoutTestCase.testNull(X)
at java.lang.reflect.Method.invoke(X)
[. .]
]]></log4j:throwable>
</log4j:event>
```

Figure 1.1: XML logmessage

This sample message includes the above mentioned basic elements of a log message such as a *timestamp*, a *severity* record (here called level), an element containing a Java stacktrace and the message itself.

As seen in the example, log messages, or at least the logged message itself are typically plain-text messages. This facilitates reading such messages by humans and transferring them between inhomogeneous systems. However, if the performance of the logging system is a main requirement, log messages may also binary encoded and converted to human readable output by using dedicated log viewers in an intermediate step. Especially accounting information is often logged in the latter format, since it serves mainly as input for statistical applications rather than for human inspection.

Although there are widely adopted standards for formatting log messages like the Apache Common Log Format [com06] for web servers or the Unix Syslog format [Ger05], there are no or little efforts to create a standard for the log message format or encoding. The reason lies in the different requirements a software system imposes on the underlying log implementation.

For example, in heavily multi-threaded systems it is desirable to add some kind of identification for the current thread (like a Thread-ID or name) to the above mentioned basic elements of a log message. On multi-user systems a requirement might be to include the user name of the currently logged in user to the log.

To overcome these requirements nowadays many logging systems use XML as data format, since standardized procedures and mature and therefore robust appli-

cations to handle, transform and display XML contents are available on nearly every system platform. An example which demonstrates the advantages of using XML as data format is the possibility to utilize XSLT, which may be used to transform a specific log format to a well known format such as HTML. Furthermore native XML databases could be used in order to provide convenient storage and retrieval facilities for the generated log data.

1.2.2 Transfer of Log Messages

Once a logging message is generated, either by the logging application itself or by a dedicated logging library, it typically gets transferred from its generating source to a defined destination (sometimes denoted as “message sink”). Modern logging systems, taking advantage of the operating systems communication facilities, provide a huge amount of possible processing and transfer modes, so in this chapter I would like to outline the path a log message takes from its creation to its final destination.

Basically current systems differentiate between the following sources of logging data:

Operating System As the operating system is responsible for managing hardware resources such as memory, disk space, and input/output devices, errors and status information is essential for debugging an operating system failure or its health status. Therefore a logging subsystem is integrated in nearly every modern operating system. Examples of such systems are the BSD syslog-daemon or the Windows EventLog (see chapter B, p.74).

The logging subsystem is generally accessible through an application programming interface, which provides abstraction of the underlying logging framework and allows configuration of the whole logging process in a central place. Typically such operating system integrated logging subsystems are configured through a system administration console or by using dedicated logging configuration APIs.

Application Besides the operating system also applications may produce logging data. Typically such applications use the logging infrastructure provided by the operating system by using the appropriate application programming interface, but often a simple file based logging is implemented. In order to separate the logging process from the underlying operating system foundation, in most cases the logging process itself is not handled by the application, but delegated to a logging library such as for example log4j for Java applications or syslog¹ in Unix environments. These logging libraries handle the needed calls to the OS-provided logging facilities or provide own implementations to store and process the logged data.

This approach facilitates platform dependent application design and unifies the whole logging process. Furthermore such proceeding allows configuration of the logging process without touching the application code.

¹note that the name *syslog* denotes both the protocol and the logging library

External In distributed systems also logging events from outside the local system may be regarded as a source for log messages. For example the previously mentioned log4j and Windows EventLog (among others) allow collecting events from arbitrary hosts over a network and injecting them to the local logging infrastructure. Typically the logging API provides the desired location abstraction, so that logging to a remote facility is handled (at least from the application point of view) exactly as logging to a local repository.

The ability to process log messages from external sources makes particularly sense if not all of the involved systems in a distributed environment are equipped with a dedicated storage to file log events, or if a central log repository is desired. In this case, such systems forward their log messages to a dedicated node in the network which processes and stores them.

Once a log message is received by the logging subsystem it typically gets processed, filtered and forwarded to some output module which writes the message to disk, to a network socket or other communication channels.

Since modern operating systems offer a vast amount of communication facilities, the possible destinations of a log message depend on the capabilities of the logging infrastructure. For example, following destinations might be conceivable:

File logging to a file is probably the simplest way to store log messages persistently. Nearly all currently existing logging frameworks include this feature and applications not using any logging framework or library typically offer this destination as default.

In fact on Unix systems (and on many other operating systems) logging to a file is built in into the command interpreter, usually called “shell”. After invoking a command, the shell provides 3 predefined data streams to the started program, called “stdout”, “stdin” and “stderr”. “stdout” and “stdin” are used to output and respectively input data from and to the program and “stderr” may be used to emit error or debugging information. With this separation of normal program output from error or status messages and the ability to redirect all of these three streams to files or other programs, this feature forms a basic implementation of a logging system. By combining these basic facilities even more sophisticated demands may be satisfied such as for example logging to remote locations or processing the data by using filters prior the actual data gets stored onto disk.

IPC instead of using files as destination for log files, also operating system provided IPC² facilities like named pipes or domain sockets may be used to transfer received log messages to other processes. These processes are typically dedicated logging systems, acting as gateways to a central logging repository. Furthermore, such processes might be used in order to store incoming messages to databases, files or other storage.

Typically however, IPC mechanisms are used for transferring log data from an logging library to a dedicated processing application only, therefore not representing a full-fledged log message destination. On Unix systems for example, log messages are transferred from the application to the syslog daemon via a dedicated Unix

²IPC: Inter Process Communication

1 Introduction

domain socket³. The syslog daemon then decides, on the basis of his configuration, where to store or transfer the received message.

Network For transferring log messages outside of the local system, logging frameworks typically support network connections. The actual used transport and encapsulation formats typically depends on the designated requirements, therefore many possible transport and encapsulation formats for transferring log messages may be used. Syslog for example, uses UDP packets to transmit logging information to remote systems, the Windows Eventlog uses DCOM as default.

Looking at the possible destinations it is possible to further categorize them to *consuming destinations*, which receive log messages and store them without forwarding (thus being a final destination for a log message), or *forwarding destinations*, which receive log messages, eventually process and forward them to other logging systems. For example file or database-based destinations are typically consuming destinations as they are typically used in order to safely save the log-data onto persistent storage. In this case, after the log message is successfully saved, it gets typically *not* forwarded to another system. Network-based destinations on the other hand are typically forwarding destinations, as they are only used to forward messages instead of storing them.

Combining and stacking various log message sources and destinations together allows creating sophisticated logging systems, as shown in an example in figure 1.2, where a message sent from an application is passed to a logging framework, which feeds the message to a log file and a network socket and passes the message to another system where it gets logged through the logging subsystem to a file.

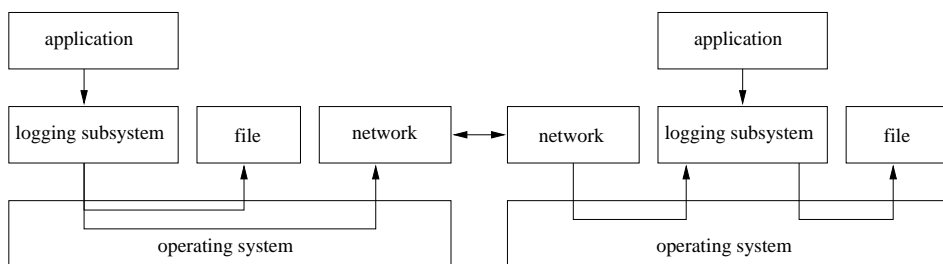


Figure 1.2: Example transfer of a log message

1.3 Introduction to Log Management

In today's computer systems nearly all components produce some kind of log data. The range spans from the operating system itself to system software and user applications to external hardware components like network equipment or storage subsystems. As seen in the previous chapter, the possibilities to transfer a log messages from its source to a destination are manifold. Thus, implementing a system-wide policy, covering all log-related matters is an ambitious task - especially in large,

³typically `/dev/log`

inhomogeneous systems. In order to facilitate such requirements and provide centralized management of all logged data, log management software was introduced. In the following chapters the requirements, the technical overview and usage examples of log management systems are outlined and advantages and drawbacks are discussed.

1.3.1 Requirements

Since log events are typically generated at a continuous rate, storing them reliably and processing the generated data in an acceptable time can be a challenging task. Especially the huge amount of generated data imposes the need for sophisticated storage systems and analyzing tools. Legislative requirements may impose specific proceedings regarding log data retention and security obligations as stated in [KS06]. Debug or statistical use cases or backup policies may require a large time horizon to be covered by log files.

Furthermore, in inhomogeneous or distributed environments log messages are typically not generated and sent from a single system, but possibly from many incompatible systems, thus making centralized log retention and analysis extensive. Such scenarios imply the requirement that a log management system should support different input formats and sources, preferably configurable at runtime.

To feed statistical applications or intrusion detection systems, the data stored in a log management system needs to be accessible in a standardized way. Searching and filtering specific log messages or trails of log messages should be possible as well as exporting them to arbitrary formats.

In security sensible configurations the reception, transmission and processing of log messages should be secured against wiretapping or other unauthorized access. Moreover, the logged data should be stored in a tamper proof way, such as on read-only media or other highly secured media.

As a log management system typically acts as central hub for all log-related issues, failures or overload situations should be handled without impact for the whole infrastructure. Furthermore, loss of log messages may imply serious problems, particularly in security sensitive environments, where authentication logs are used to track (possibly unauthorized) system access. To overcome this threats, log management systems are typically implemented as distributed, redundant services.

Applications handling these requisites are referred as *log management systems* and given that all those requirements are hardly satisfiable by ordinary, monolithic systems, nowadays such systems are build on a modular basis, where modules are easily interchangeable or customizable, preferably at runtime.

The technical architecture and specifications of such systems is outlined more detailed in chapter 2.

1.3.2 Log Policy Definition

A fundamental problem of log management is to find a way to balance the quantity of generated log data with the available resources, such as storage and processing power. This implies that some compromise must be taken, in order to record only relevant information but in the same time store enough arbitrary data to be able

to effectively feed use cases which may require more specific information, which is typically not considered important in an controlled operation. Especially debugging and optimizing use cases are often possible only with very detailed log data.

In this chapter I would like to introduce methods for defining a log policy and commonly used practices for handling the above illustrated dynamic logging demands.

Although a log policy is highly dependent on the actual system requirements, and thus needs to be specifically adapted, the following basic phases of logging should be considered while designing a log policy (as stated in [KS06]):

Log Generation: Defining where, how and which log data is generated forms the most challenging part of log policy definition, because this step of the log policy creation process defines for the most part the information yield contained in the log message itself and the information gain that may be obtained by analyzing series of correlated log messages. Particularly defining at which events or rate a log message has to be generated and which information needs to be included affects the effectiveness of further problem analysis or debugging activities. For example, triggering log message generation only on a small amount of possible events may cause loss of debug possibilities obtained through log message inspection, because of non-availability of arbitrary, debug or informational type of messages. On the other hand, logging as much as possible data may implicate information overload, thus constricting effective and goal oriented log analysis.

As seen in this example, this step of policy definition is highly dependent on the particular use case, thus it is highly recommended for a log message system to permit modification of the log policy at runtime, in order to adapt quickly to changed requirements.

This requirement is especially true if the log management system should be able to react on external events, in order to dynamically adapt the current logging level. Such level changes could be for example, triggered by network intrusion detection systems, which typically analyze traffic anomalies to adapt the logging level accordingly. A increase of suspicious traffic could for example trigger an increase of the security log level, in order to record detailed information about this exceptional situation for later analysis. After the decay of the traffic anomaly, the log level could be re-adapted by the system to a specified standard level.

Log Transmission: In distributed systems, after a log message is generated it typically gets transferred to a centralized log management system for further processing. By defining a policy on how a log message is transferred and which hosts or devices on a network take part of the logging system, a consistent and reliable log infrastructure can be achieved. Furthermore it is advisable to specify the frequency at which log messages are transferred: Near real-time log message transfer for systems where the ability to quickly react on certain system events is an absolute must, or transmission of logged data in batches on systems where logging is primarily used as source for statistical data or other non time-critical purposes.

In security sensitive environments it is desirable to encrypt log message delivery or even have the transfer of log messages separated from other data transfers by

moving it from the corporate LAN to a separate, dedicated logging network. Such a logging infrastructure is especially useful in cases where the logging system is used to monitor the corporate LAN and failure of the such should not interfere with the logging system.

Highly secure environments may furthermore require to store all generated log messages in a special intermediate repository before they are transmitted to a remote location. By storing the original messages it is possible to witness correct transmission of log messages by comparing the transmitted data with the stored one. Using the same method malicious manipulation during the log transmission is easily detectable.

Log Storage: Once the message is transferred to its destination it gets typically stored in a database or other storage facility for later inspection. To assure log data availability and at the same time limit the amount of long-term stored data, a log storage policy defines how long which type of data needs to be stored and how the log data management process is implemented. Therefore the log storage policy defines how often logs should be rotated as well as the security constraints applied to the stored log data. A storage policy should furthermore include detailed statements on how long-term data, such as data outside the current log rotation scheme is stored in order to achieve consistent records of logged events even on a longer time-horizon.

Furthermore, a log policy should contain proceedings and instructions on how integrity and availability on the whole logging system could be established and maintained and how failures of the system are handled.

As log data often contains vital and security related information this requirement has to be taken seriously. Especially in distributed systems, where log messages are transferred among systems and on probably unreliable network connections, loss of log messages needs to be avoided and mechanisms to detect such failures need to be implemented. The Unix syslog daemon [Lon01] for example, in its standard configuration uses UDP, a connectionless and “best effort” protocol to transfer messages over network links. Since the design of the protocol does not guarantees a correct transmission of log data or acknowledge of reception, it is not very adequate for transferring audit logs or other security aware log data, whose successful reception must be guaranteed.

Just like the transmission of log data across network links also the process of writing logged data to the storage system must be covered by the log policy. System failures such as service interruption during the storage procedure or other unforeseen events need to be considered during the log policy creation. More general requirements for the storage subsystem are discussed on chapter 2.1.3.

Another important aspect to take into consideration is that a log policy should also include detailed information about what is *not* logged by the given specifications. This information is typically helpful when the systems reliability needs to be revised. Only by defining also events not covered by the log policy it is possible to decide if a event was lost due to an error or bug in the logging system or was not logged deliberately.

Because the previously mentioned three phases of policy definition cover only

general requirements, an actual log policy should include more detailed rules. An example of which policy elements should be included in a log policy suitable for security sensitive environments will be discussed consecutively and best practices will be stated:

Read/Write Access of a Resource: since unauthorized access to data can be typically classified as read or write access, such events need to be included in the log policy. Even in environments where logging is deployed for statistical and debugging purposes only, read and write accesses to a resource or data can be a valuable source of information in case of a service misuse.

Deletion of Data: logging of data deletion is clearly a further, important access pattern to be included in a security sensitive log policy. Without logging deletion of data or resources it is nearly impossible to track down deliberately destruction of data and system attacks as well as detecting obfuscation of successful system break-in attempts.

Like read and write access, also the notification of file deletion needs to be supported by the used operating system. Windows, and its NTFS file system allows very fine-granular tracking of file system based events, configurable on a per user basis. On Unix and Linux-based systems, daemons or libraries like fam (file alternation monitor) [KM06] allow user mode programs to be notified on various file system related events.

Modification of Resource Properties: Since resources may have an arbitrary number of properties, for example permissions, security labels or ownership, the modification of those properties needs to be covered in the log policy in order to track unauthorized access to this type of meta data.

Network Activities: Although due to the amount of information it is nearly impossible to log all transmitted data, at least connection attempts, connection establishments, aborts and failures need to be logged.

Authentication Events: All authorization attempts should be logged. The log message should include the time, the result such as for example “success” or “failure”, the object to which the authentication was attempted as well as the user credentials. It is important that also advanced events such as log out events, password changes, etc. are logged, since they are required to track the actual service access patterns or possible service abuse.

Administrative Access: In addition to data access by users, also accesses effected by administrators should be logged in order to identify privilege misuse or escalation. Furthermore, administrative access to user data, access control lists or logging and system configuration should be considered in the log policy definition. Of course logging administrative access requires strict security permissions on the logged data itself, preventing modifications of the logged data by system administrators, who’s privileges are typically much elevated in respect to ordinary users.

While log policies for simpler systems typically include some or all of the proposed entries, the design of a log policy for complex, distributed systems is considerably more extensive. The problem in such environments is how to specify a log policy which does not include the individual systems exclusively, but the distributed system as a whole. Actually, the functionality of such system is not defined by the functionality of a single system component, but by the combination and collaboration of the concerned systems. Therefore, the definition of which events need to be covered by the log policy has to be extended by events generated by the *interaction* of the interconnected system components. However, the process of identifying those events is not an easy task, since often side-effects caused by the interaction of interconnected systems are hardly predictable. Therefore, besides regarding certain notable events as important, especially all unforeseeable events should be considered as events with elevated importance and logged accordingly.

Decision Context: As mentioned in the previous chapter, most log management systems use discrete, hierarchical log levels in order to categorize events regarding their importance. This level is typically set on each log message by the log-emitting application either explicitly or by calling the appropriate function of the used logging library.

However in large, distributed environments, the decision of the application running on an individual system, which log level to set for a specific event is often not relevant for the system as a whole. This results from the fact that in a distributed environment, the individual node has generally only limited knowledge of the surrounding environment⁴. To clarify this statement I would like to give the following example:

Suppose a computing cluster consisting of a number of slave nodes and a master node which supervises the cluster functionality and distributes the workload across the slave nodes. In this environment, a log message of a slave node, alerting a disk failure is at the first moment only relevant to the master node, in order to exclude the node from further workload allocation. Although critical for the node itself, as long as the cluster itself works as expected, this failure message is not critical to the cluster functionality. Therefore, the master node typically may not forward the original log message with level “fatal” or even “emergency” to a superordinate systems console, but rather a more appropriate log message of “warning” level.

In such environments, the original source of the log message is typically not able to select an appropriate log level because this decision would require some knowledge about the surrounding environment the node is embedded in. More generally, in order to make an adequate decision about the importance of a log message, the *context* of the occurred event needs to be considered. Since an individual subcomponent of a complex system usually has no command of such context, the decision is typically assigned to a superordinate system. This system, for example a log management system, processes the received log messages and tries to correlate them by using predefined or dynamically generated rules. This a-priori, respectively dynamically *learned* knowledge, together with information about the system architecture

⁴ Actually, such behavior is highly desired in loosely coupled architectures, where each application acts in its strictly defined domain.

and current operational properties forms the mentioned context. By integrating this context into the decision process, the system is able to separate unimportant messages from important ones and react accordingly. It is therefore advisable for future log management systems to include extensive support for correlation algorithms and message processing facilities as outlined in chapter 5.

It is however important to note, that advanced event processing algorithms such as event correlation should be used preferably to assist log-related decision processes rather than replacing them. The aggregation of related events could for example lead to loss of information, if the correlation algorithm fails in identifying the real importance of the processed events. Therefore, *all* received log messages should be saved for later reference, regardless of their calculated importance.

Non-Functional Requirements: After discussing useful considerations for designing a log policy, it should be pointed out that a log policy needs to include also non-functional requirements. An important example of such requirement is the specification what actions need to be taken, if the log management system fails:

- how should the failure be alerted?
- are other systems affected by a failure of the logging system?
- is the logging system of vital importance?

An important question here is, how important the logging system is for the whole infrastructure. If a logging system is used for accounting or monitoring purposes, a failure of such a system may be fatal and therefore it is advisable to specify the further proceeding in such a situation. For example in environments, where accounting data provided by the logging implementation is essential, a failure of the logging subsystem might imply an immediate shutdown of other, concerned systems in order to avoid loss of data. If the logging system is used as data source for monitoring or alerting purposes the log policy must define specific directives how to proceed in case of failures.

Last but not least, a log policy has to be feasible. Since logging typically contains valuable information and in the most cases presents the only source of system and debugging information, the logging infrastructure and especially the administrative staff must be able to handle the volume of generated data. The most detailed and elaborated log policy is useless, if the system fails in extracting the needed information out of the logged data or provide sustained reliability in a given time span.

Therefore it is advisable to *test* the specified log policy for being feasible at high system loads. Also the log analyzing algorithms should be tested whether they provide the needed information for debugging or forensic purposes.

2 Log Management Systems: Current Concepts

Logging of system events was an requirement for computer systems since their beginnings. While the first systems simply sent their log data to an attached printer, nowadays almost every operating system or hardware equipment has a built-in logging service respectively the possibility to redirect log messages to a centralized logging server. However, the large amount of data generated by those systems makes it very cumbersome to process and analyze it for specific patterns. Moreover, with the volume growth of log data, storing and managing those log data becomes a demanding task. This is for example the case, when gap-less recordings of all events for a defined time span are desired, for example for failure analysis or for statistical purposes.

Where in former times rather simple writing of received events to a log file was adequate, nowadays busy web servers or other high-performance systems generate logging data in the magnitude of hundreds of gigabytes a day. Storing and managing such amounts of data clearly requires more sophisticated architectures and algorithms than simple file-based storage. These kind of systems, providing means to store, manage and analyze the logged data are typically referred to as *log management systems*.

After outlining their basic structure and discussing the detailed design of the various system components, in this chapter algorithms for processing and visualizing large amounts of log data are presented.

2.1 Architecture

Although the specific technical architecture of a log management system has to be adapted to specific requirements, the basic structure of a log management system is layered in three tiers:

- input layer
- processing layer
- output layer

All log messages are received by the *input layer*, processed through the *processing layer*, and sent to the storage layer or other logging systems by the *output layer*. Each of those layers may be split up in an arbitrarily number of modules, allowing to stack different types of modules together, therefore allowing the log management system to support a large number of different output, input and processing formats in one single application.

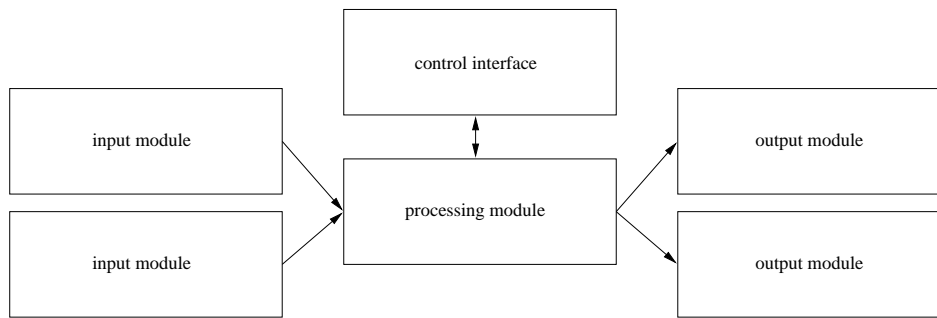


Figure 2.1: Example log management system

An example of the structure of a log management system is shown in figure 2.1.

As the modules are loosely coupled to each other it is feasible to move some or all of them to separate machines, thus forming a distributed system. This approach is often considered when availability and scalability are one of the key requirements. Furthermore, breaking up the functionality of each layer into modules greatly improves extensibility, since new features are easily integrable by simply adding a new module to the system. Such architecture is also more adaptable to different system environments and usually more secure than building monolithic applications.

The wiring and configuration of these modules is typically handled by the core logic of the log management framework and is typically adaptable at runtime. It is however not uncommon to include log management systems or parts of them into other applications, as for example, monitoring systems or other log related applications. In this case, the wiring of the different modules is typically delegated to the surrounding infrastructure, in order to allow centralized configuration of the system and seamless integration of existing configuration frameworks.

In the next sections, the detailed architecture of the presented basic modules of a log management system will be discussed:

2.1.1 Input Layer

The input layer is responsible for receiving log messages and transforming them to a defined format for further processing by the processing layer. Typically the input layer is split up in modules, usually implemented as runtime loadable modules.

Such approach allows adapting the log management system to legacy systems and building location independent logging infrastructures by allowing reception of log messages over network links. It is important to note that input modules are not explicitly bound to network or IPC type of communication, but may be also used to import data from databases or simple log files. This fact is particularly interesting when a consolidation of traditional file or database-based logging systems to a centralized log management system is desired. In this case for example, an input module may provide support for watching a specific file, transforming appended lines into standardized log messages acceptable by the processing layer¹. The type of input module typically also defines the time of processing of the received log

¹in fact, the well known Java log4j logging system provides this type of input module

messages. While by using modules such as, for example network-based modules, a log message gets immediately processed immediately by the processing layer, file or database based modules may be used to import and log messages recorded at a previous point in time.

2.1.2 Processing Layer

The processing subsystem implements the core functionality of a log management system. From a architectural point of view, the processing layer is typically built upon two main modules, the *processing core* and the *rule store*. Log messages received by one of the configured input modules are processed by the processing core which filters them against rules contained in the rule store. The result of the filter process decides the further routing of the log message to the different output modules.

The rules which are used to process incoming log messages are typically expressions containing matching and comparative components and evaluate to a boolean result. The processing core uses the matching component of a rule to extract the needed information out of a log message property and compares this data with the given comparison expression. The result, a boolean value, is used to decide which of the configured actions should be executed next. The possible actions may be classified in one of the following categories:

Forward Action: The log message gets filtered and forwarded to the output modules.

New Rule: The result of the matching process is a new rule.

Trigger: The result triggers an action, such as displaying an alert or running an external program.

In addition to the initial configuration, the rule store typically supports reconfiguration at runtime. The new configuration might be supplied through a dedicated configuration interface, by providing a file containing the new configuration or by an above mentioned *new rule* action. The “new rule” actions enable the log management system to dynamically adapt the active rule set to the given environment. For example, if a logging source tends to overflow the log management system with an disproportionate amount of messages, the processing core could create a new limiting rule on the fly, in order to filter out the flooding source or rate limit the reception of log messages. Typically this dynamic rule creation process needs to be configured beforehand. For example, a typical rule for triggering a counteractive action² might be “limit a host if amount of messages per second > 1000”. This approach however requires that all possible events, on which the system should react on need to specified manually. In larger environments however, this task is nearly impossible or at least tedious and error-prone because of large amount of possible unexpected events that arise from the huge amount of participating systems and their interactions. Therefore, many large log management systems use self learning algorithms such as neuronal networks to continually watch and learn the regular

²not to be confused with the rules in the rule store

system state and engage evasive actions by creating specific rules on unexpected situations. It is important to note that such self-learning algorithms are typically not allowed to directly intervene on a running system, but are used to aid the administrators in creating an initial rule-set or adapt a predefined rule-set to a running system.

Besides the mentioned features, the processing core may also be responsible for identifying and reducing redundant data, thus limiting the amount of logged data. For further information about this feature please refer to chapter 2.2.

Since the processing layer implements the core functionality of the log management system it typically implements the interface for administrative control. Through this interface the rule store can be managed as well as the configuration of the input and output layer. This includes also the initial configuration, containing a basic rule set and module wiring.

While the processing layer can be typically found in nearly all log management systems it is possible to implement so called proxy systems, which contain only a basic, stripped-down processing layer. Such systems are typically used to convert between different logging systems or formats. They are built by an input layer and an output layer, assisted by a simple processing layer, which is exclusively used to forward messages between input and output layer without applying any further processing.

2.1.3 Output Layer

Since retrieving and particularly storing received messages is the most important requirement, a reliable storage subsystem is one of the basic building blocks of modern log management systems. Typically such systems use database back-ends for storing received log messages, but on devices with limited computing or storage resources also simple file-based approaches are feasible.

Generally the storage system should provide the following features:

ACID Property: This acronym stands for “Atomicity, Consistency, Isolation, and Durability” and is a key requirement for reliable database systems [ISO98]. Nowadays almost every database system implements this requirement, which guarantees that the stored data is safely written on the physical storage even in critical situations such as power outage or other unpredictable failures. Therefore, if for example a log management system provides only simple redirection of log messages to a file or other basic storage capabilities, special precautions should be considered to ensure that all four requirements of the “ACID” property are fulfilled.

Log Rotation: As log messages are generally generated in a continuous way, the storage subsystem should be able to handle rotation of the logged data on a predefined scheme. In order to limit the amount of data a search for specific log messages needs to inspect, the log data is typically stored in a time-hierarchical storage. Typically, a specific amount of messages, for example, all messages recorded during 24 hours are recorded to a single file, thus limiting the amount of messages contained in a single file. Therefore, searching for an event occurred in a specific day requires only accessing the log file for that

given day, without accessing other logs, therefore resulting in a massive speed gain. It is important to note that such simple log file rotation needs to ensure that no incoming log data gets discarded during the actual log file rotation process. This requirement is typically assured by implementing a buffer either in the log management core itself or in the output layer.

This principle, of storing a specific amount of log messages in separated storages can be further refined by introducing a storage hierarchy, where freshly recorded data is stored on fast media and not recently accessed data is automatically transferred to a long-term storage area. Typically such functionality is provided by the operating system, but it is feasible to implement it into the log management system in order to provide a platform independent feature-set.

Support for Indexing: Since many management tasks, most notably log extraction or analysis requires fast access to the stored log messages, the storage subsystem should be able to handle search and filter operations in a reasonable time. Typically such requirement is fulfilled by implementing indexing. However, while using database systems as back-end for log data storage this functionality is implicitly provided, simpler forms for storing log messages like the mentioned file-based approach typically do not provide this feature. In this case it might be advantageous to implement the previously mentioned log rotation in order to limit the amount of data that needs to be considered during a search operation.

Redundancy: Like every other component, the storage subsystem should support redundancy and fault tolerance techniques such as distributing storage to multiple physical locations or cluster architectures. Such architectures do not only increase the availability of the stored data, but as a side-effect often also the data throughput during search and filter operations, since multiple physical storage devices can be accessed simultaneously during read requests.

Support for Access Control: As the storage of log management systems potentially includes many security sensible entries, such as authentication information, service details or other valuable information for unauthorized parties, controlling the access to the logged data is a essential requirement of a log management system. Typically access control can be implemented by the operating system if simple file-based storage is used, or by the log management system itself, respectively the database back-end used by the output module. The latter method yields the advantage of a more fine granular access control, typically enforced as far as to a single log data entry.

Compression Support: Last but not least it is desirable that the storage subsystem supports compression of the stored data, especially regarding the fact that most logged data is highly redundant, thus being well appropriate for compression. The compression algorithm should support transparent read access to the compressed data store, because this way, typical log management actions like searching for data or generating reports from the logged

data might be implemented without decompressing the whole data store. Regarding this requirement it is feasible to use operating system provided file system compression, since being directly integrated into the operating system this approach provides transparent compression and thus simplifies the overall design of the various output modules of a log management system. However, when storage of log messages to database based output modules is used, this requirement might be considered non-relevant, since database systems typically use their own highly optimized storage engines, providing performant and space optimized data storage making explicit compression unnecessary.

Reviewing the mentioned requirements for the storage layer, it is clear that using a database system as storage for the logged data is the most evident solution for building a reliable and fast storage core. Database systems typically fulfill nearly all stated requirements in their default implementations, so that the complexity of the log management system and especially the storage layer can be reduced in a considerable degree.

If the usage of a database for log management is inappropriate on a storage and CPU-power restricted system, the output system may be implemented by a network-based output module, which forwards the log data to a more powerful system acting as database back-end.

In distributed environments, the output layer is further used for transferring log messages across network links to other log management systems or intermediate systems, such as for example, proxies or other application layer based network equipment.

Since reliability and availability needs to be guaranteed on the output system on the whole, also the network subsystem of the output layer should be designed with that requirements in mind. It is advisable therefore to design the system from the ground-up with substantial error tolerance, since error-free transmission of data cannot be assured by many network links. Acknowledging of the transmitted data by the network peer, re-transmission of failed data, buffering and other methods used to overcome unreliable network links, such as described in [TS02] have to be considered when designing the output layer network subsystem. Often, the right choice of the transport protocol, such as, for example, favoring TCP connections over UDP datagrams on IP networks eases considerably the design of a reliable log management system architecture.

2.2 Message Processing

As seen in chapter 2.1.2, the processing core implements the main functionality of a log management system. Since log management systems are not exclusively used to simply store all received log messages but, as the name implies, to manage and process the collected information for further use, this chapter outlines the details of the processing procedure itself. Furthermore, commonly used methods and approaches for filtering and handling huge amounts of data will be identified.

The steps for processing a log message are typically the following:

- interpretation

- categorization
- normalization
- reduction
- execution

Typically, the processing starts after the log message has been received by one of the input modules. In this first step, the contents of the log message gets *interpreted* and converted to an internal representation. This approach enables processing messages from inhomogeneous sources in a consistent manner. To accomplish this requirement, the design of the internal representation of the log data should be adaptable to new requirements without reimplementing the entire processing core. Such behavior is often achieved by designing a basic data structure, containing fundamental properties needed to model a log message and designing the processing core and all other system components to be able to handle also enhanced versions of this basic structure³.

To handle large amounts of concurrent data streams, the processing logic may be enhanced by a buffering system, used to handle overload situations. This buffer stage may be further used to filter out sequences of uniform messages before the actual processing step, thus effectively rate limiting floods of similar messages and avoid the burden of processing them one by one.

The next step, *categorization*, consists of matching the received log message (or more specifically, the internal representation of it) against the rules contained in the rule store, thus deciding which further processing steps are to be executed, whereas a rule contains typically the following basic components:

- match
- operator
- comparison

The *match* component is used to identify which property or meta data of the current log message is going to be compared using the *operator* with the *comparison*.

The *match* is typically a string matching or regular expression, but can be virtually any expression whose result is comparable with another discrete value. Examples of such matches are the expressions in table 2.1.

It is important to note, that the matching may be performed also against arbitrary values (as seen in table 2.1), not necessarily contained in the log message itself. Such values, quoted in the *comparison* component, can be for example meta data, such as the time of reception, the amount of received messages per time unit or other, user defined values, or mathematical expressions using those properties. An example of the matching process is given in figure 2.2, where the match defines that property1 (the level) should be matched. This value, “debug” is compared

³this approach is easily implementable by using object oriented, inheritance capable programming languages

using the comparison operator “>” to match the fixed value “warning”. As outlined in chapter 1.2.1, by using hierarchy of discrete values, the property “level” can be compared using a relational operator, in this example resulting in the boolean operator *false*.

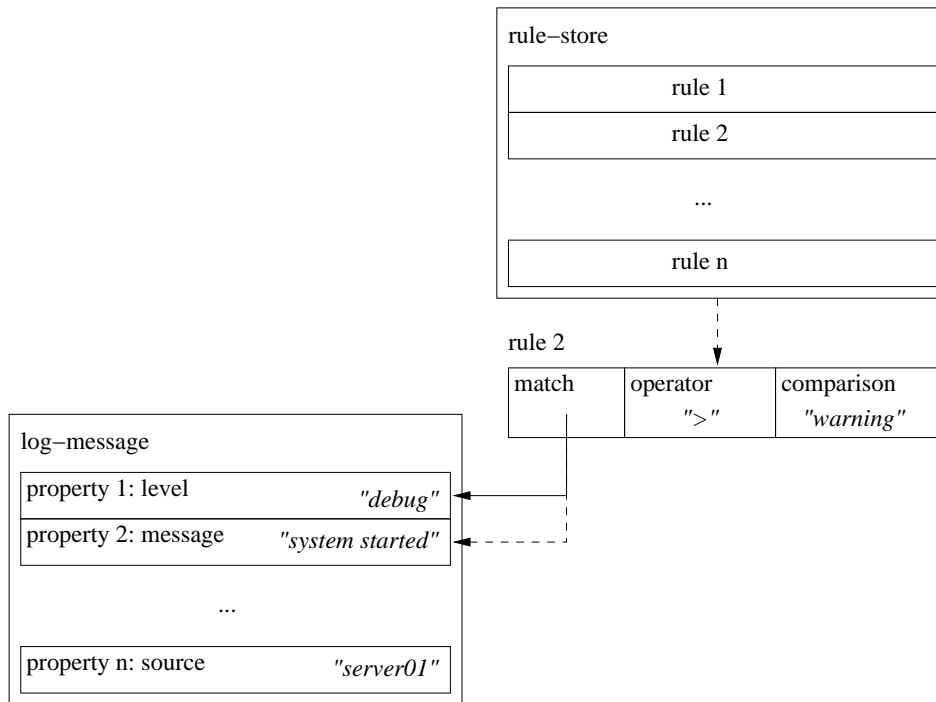


Figure 2.2: Matching process example

To allow forming more extensive expressions, rules are typically chained together using boolean operators, thus for example expressing dependencies among rules may be achieved using expressions like:

$$\text{rule}_1 \wedge \text{rule}_2 \vee \text{rule}_3 \dots \text{rule}_n$$

Using such expressions actually transforms the rule list into a dependency tree, typically implemented in the processing core as boolean decision tree. A log message is therefore processed by traversing the decision tree, finally resulting in a boolean value.

EXPRESSION	DESCRIPTION
PRI	match the priority field
message size	meta data match: size of the log message
time() - receive-time()	match using external meta-data and mathematical operations

Table 2.1: Examples of match expressions

This boolean value typically decides the final destination (or destinations) for the message as described in chapter 2.1.2.

However, many log management systems do further processing prior executing the calculated action, mainly to reduce the amount of triggered actions. For example, using a ring buffer or history of former processed messages, it is possible to detect sequences of uniform messages, thus making it feasible to merge those superfluous messages into a single, representative message, typically extended with the information about the amount of suppressed messages. The algorithms used to reduce the amount of messages are of course not limited to simple ones, identifying solely equal messages but may also include sophisticated algorithms based on fuzzy pattern matching or statistical approaches. This processing step is outlined in more depth at chapter 2.3.

As the final destination may be anything from triggering an action, such as alerting the administrator or forwarding the message to the storage layer, this wiring between the result of the processing step and the to be considered action needs to be configured by the administrator. Typically, such configuration is implemented as simple mapping between the result and the actions, like for example:

```
if (rule1 and rule2) then log message to file "alert.log"
```

Of course it is feasible to configure more than one action per result. Diverting log messages to multiple destinations, for example displaying the message on the administrator screen as well as storing it to the storage system for later reference, represents an often used configuration.

2.3 Log Data Analysis

2.3.1 Introduction

Log data being simply stored on storage subsystems like files or databases provides no real benefit, besides the fact of its reliable storage. Extracting information out of the logged data however, represents one of the main benefits of a log management system. Therefore, a log management system typically provides an interface for analyzing the logged data or filter and export messages for further processing.

2.3.2 Use Cases

Since logging of application or system events generates huge amounts of data, as outlined in [Ada02], the need for efficient management and analysis of the recorded data is evident. Typically log analysis is used for example in the following use cases:

Forensic Analysis: Forensic analysis is a typical use case which needs detailed log data. After a break-in attempt or other malicious activity, system administrators or security responsables typically need detailed logging information in order to narrow down the vulnerability that lead to the possible successful break-in attempt. In this situation, a centralized log management system is typically the main source of information, since typically all security related

events, such as authentication attempts or service logs are collected by such systems.

Debugging: Another well known use case for log data analysis is debugging. Nearly all applications allow setting the amount of logged data to a value that allows tracing detailed application events. Setting the log-level to a specific value is typically possible on runtime, thus avoiding a shutdown of the application just for changing it, which is often not feasible on mission critical environments.

Typically, a logging framework provides one or more specific log-levels (typically called *debug* or *trace*) in order to separate debugging data from ordinary logging data. (An example can be found in table 1.1).

Anomaly Detection: Besides the forensic analysis, another scope for log analysis is anomaly detection. Anomaly detection basically describes the ability to react on uncommon events. Typically intrusion detection systems use the data provided by the logging framework in order to detect malicious or abnormal access patterns. In consequence such activities can be alerted to the administrative staff, providing early detection of unintentional accesses to security sensitive systems. This use case is however not restricted to monitoring security related events, it is feasible, for example, to use anomaly detection also for monitoring system load or service availability in order to react before such events might lead to major failures.

Statistical Analysis: Since log management systems store the logged data for substantial time spans, the contained information provides an excellent base for statistical evaluations. In fact, especially log data gathered from application servers such as, for example Web servers or mail servers is often used for statistical purposes. Such statistics provide valuable input for upgrade planings by providing data like system utilization over a given time. Particularly Web server logs are typically processed in order to gain statistical information about the users path throughout a web site. By analyzing this data, a site could for example be optimized in regard to its user interface, or from the technical point of view, by placing often accessed items on dedicated servers or caching systems in order to increase the response time.

Reporting and Accounting: Last but not least, log data analysis provides also valuable information for reporting and accounting purposes. For example authentication events might provide data for user or usage based billing systems. Furthermore by extracting data from logs it is possible to generate reports providing information about usage, access patterns, user movements throughout applications such as web applications or other data needed for charting or reporting.

As seen from the above mentioned use cases, there is a lot information contained in application or system logs. However, filter only relevant information out of huge amounts of data hits on many problems. In the next chapter I would like to outline some algorithms and techniques used to effectively analyze large amounts of logged data.

2.3.3 Analysis Algorithms

Typically, algorithms and systems used to extract information from the logged data pool can be categorized whether they use contextual information or not and by the time, the actual extraction process takes place:

Contextless: Contextless algorithms are typically simple search algorithms. In log management systems such algorithms are used to filter certain log events based on user-defined patterns. Typically, such patterns are formed using application specific elements or by using regular expressions.

Context Aware: Context aware algorithms extend contextless algorithms with contextual information, as outlined in the following section.

Realtime: Realtime analyzing systems are typically embedded in the log management system itself. Realtime in this particular case means the received log messages are immediately forwarded to the processing core after reception by the log management system.

Offline: Offline processing systems are applications, analyzing log messages *after* they were received and stored by the log management system. Typically, such applications operate on the log file or database used by the log management system, rather than on the actual incoming log message stream.

The problem however is, how to detect and especially *specify* events that are not detectable by analyzing a single log message, but events that are the consequence of several, not necessarily consecutive messages. To detect such *meta-events*, represented by a sequence of log events, simple text matching on phrases such as “failed”, “error while” or other simple comparisons in order to detect for example a system failure are not possible. This type of analysis, trying to extract the *root cause* of a given sequence of events is denoted as *root cause analysis* and is outlined in more detail in chapter 5.3.

Event Correlation

Algorithms matching a sequence of events in order to deduce a single, representing event therefore use *contextual information* in addition to classical search algorithms. This approach is typically referred to as *event correlation*. Jakobson and Weissman define event correlation as follows:

“Event correlation is a conceptual interpretation procedure where new meaning is assigned to a set of events that happen within a predefined time interval.” [JW95]

This definition refers to one commonly used correlation mechanism, namely correlation of messages in a given timeframe.

In order to give an example of the usefulness of event correlation I would like to state an example taken from the thesis of Risto Vaarandi [Vaa05], the author of a

software called *simple event correlator* [Vaa06a]⁴. In his thesis, event correlation is used in order to monitor a network event and forward relevant events to the network administrators:

“If the linkdown event is not followed by linkup in t seconds, forward the link down event message to network technicians; otherwise generate the linkbounce event, and if more than n such events have been observed within the last t' seconds, generate the *link-quality-low* event and forward it to network technicians.” [Vaa05]

This example illustrates clearly how event correlation algorithms can be used to summarize a sequence of events into a single, meaningful message, while sorting out unimportant and often misleading events.

In this example, the event correlation system uses user-defined rules in order to filter a sequence of messages. Such *rule-based* approach is typically used, if the the processing of events needs to be formulated using simple “if” and “then” statements like in the above mentioned example. This is typically the case if the configuration of the correlation system is carried out by end-users rather than specialists. Rule-based approaches therefore have the advantage that the entire matching process is comprehensible by the user.

However, event correlation may be combined also with more advanced algorithms, such as neural or Bayes network based approaches, artificial intelligence algorithms or graph-based methods. While being typically appropriate for extensive and demanding use cases, such algorithms are typically not fully understandable by the end-user. Using such algorithms it is not always evident how and why the event correlation system filtered the given event stream. Therefore the configuration and optimization of such systems might be more difficult than rule based ones.

Besides the increased complexity, self-learning and data mining algorithms may be used in addition to rule-based approaches as outlined in [Vaa05]. For example, such algorithms may be used to generate additional rules in order to enhance user-provided ones or generate a basic set of rules in order to be further refined by the administrator. Especially, if creating an initial rule set is tedious and error-prone the latter approach might be appropriate.

It is important to note, that an important requirement for event correlation algorithms, and for log generation in general is the requirement for synchronized clocks. Especially in distributed systems a synchronized time is essentially in order to identify the timely sequence of log messages.

Expected Events

Looking at the various methods for analyzing subsequent events it is important to note that just like the ability to match an occurred event, also the possibility to detect missing or more specific, *expected* events is an important requirement for a log processing algorithm. Typically such missing events emerge from overload situations or may indicate communication interruptions, thus forming valuable information for a log management systems used to monitor a specific service or system.

⁴similar programs are logsurfer [LE04] and logsurfer+ [Tho]. Possible usage scenarios in the field of cluster log analysis using logsufer are outlined in [Pre03]

Using realtime event processing, the non-occurrence of such expected events can be used to alert the administrative staff. When used with offline scanning of stored events, for example for forensic analysis, missing events may indicate malicious alternation of timestamp mechanisms or other periodic, heartbeat-type events.

2.3.4 Log Data Visualization

Although algorithms like those discussed in chapter 2.3.3 reduce the amount of irrelevant log messages considerably, monitoring large or heavily loaded systems by analyzing log messages however, strains even the best algorithms to their limit. In this scenario, the amount of pre-filtered messages through log correlation is still too large for human inspection or analysis.

In fact, for humans it is difficult to recognize patterns or irregularities in a stream of textual data. To address this problem, many log management systems are equipped with log data visualization subsystems. These systems are used to produce a visual representation of the log database, making it easy to quickly identify and extract the needed information. Typically such tools are utilized to analyze a stream of log messages in realtime, thus providing an immediate overview of a systems state, or as analysis or forensic tools on recorded log data.

MieLog [TTKH02], for example, is an interactive tool to visually browse large amounts of log data. It was built to assist administrators in monitoring and analyzing logged data and provides an user-friendly, OpenGL based graphical user interface.

In order to process log messages effectively, MieLog uses several steps to display the collected log messages as user-friendly as possible:

Log Conversion: in the first step, each logged event gets transformed into the so called “General Log Format (GLF)”, in order to process log message with different formats in a homogeneous manner.

Frequency Information Extraction: After the conversion to the GLF format, the MieLog system extracts information about the occurrence of events. This frequency information may be accessed in the subsequent processing steps and serves to weed out multiple occurrences of identical messages.

The frequency information is further split up according the following categories:

- regarding the time
- regarding the tag
- regarding the message

This categorization together with the frequency information assists recognizing unusual events, without requiring prior knowledge of the appearance of such events. For example a massive increase of “connection failed” messages might indicate an imminent network failure.

Log Visualization: After the log conversion and the frequency extraction, MieLog is ready to display the data using a graphical user interface. To assist the administrator in the log analyzing process, the MieLog GUI is based upon two main paradigms:

- visual representation of the textual content
- integration of frequency information and user-defined rules

where the goal of the visual representation is to allow the user to quickly detect important information and identify anomalies. Highlighting keywords using high contrast colors according to frequency and displaying an outline of the textual information of the logged messages helps quickly identifying important information as outlined in figure 2.3. The sixth line on the figure can be quickly identified as outlier by looking at the generated text outline.

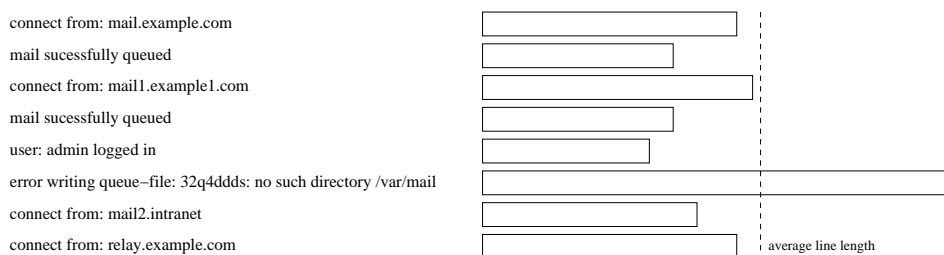


Figure 2.3: Text outline

MieLog extends this text outline representation by further adding frequency information. By filling the text outline bars with colors representing a specific frequency it is possible to display both length and frequency of a log message without overloading the visualization with irritating elements.

Furthermore, a separate screen area, the so called “tag area” displays user defined tags such as “login failure”, or “link down” and their frequency in the analyzed logs. By simply clicking on one of those tags, another display area, containing the original textual information is filtered by messages containing the selected tag. By combining multiple tags with boolean operators such as “and”, “or” and more advanced operators such as “like” and further combining this expressions with the possibility to specify a specific time span, makes it easy to filter large amounts of messages in a intuitive, thus user-friendly way.

This description stated only a selection of MieLog’s features, for a complete description of the MieLog system please refer to [TTKH02].

Another example of log data visualization is the visualization of server logs. Software packages like, for example “awstats” [Des07], W3Perl [Lau07] or others are typically used in order to periodically scan log files generated from Web servers, ftp servers or other services and generate charts and other statistical data from it. In contrast to the previously example of MieLog, the main usage of visualizing server logs is not to visually aid administrators in filtering the logged data, but rather to generate a graphical representation of statistical data. Such data could for example

include the number of visitors over a given time slot, or the ranking of the most often viewed web pages.

Applications, such as MieLog demonstrate the advantages of using data visualization in order to quickly analyze and extract the contained information out of logged data. In contrast to approaches, where log messages are solely filtered by often confusing rule specifications such as regular expressions, visual - and especially interactive systems provide more intuitive ways to filter out the wanted information. Another advantage is the immediate user feedback. For example, if a user clicks on a tag on MieLog's tag view, the system *immediately* applies the therefore edited filter rules, thus providing instant feedback. Such behavior typically enables constructing rules in a top-down manner by gradually narrowing the amount of displayed data, by adding more specific filters in each step.

It is important to note that graphical systems are not only appropriate to interactively filter logged data, but also for constructing and testing rules used to configure autonomous log analyzing systems. This approach is especially useful when specifying rules and patterns for such systems is tedious and error-prone. Therefore, many log management systems support a hybrid approach for constructing the needed rule set: In the first step, log messages are sampled from a real system. These gathered data is then filtered and analyzed by the operator using the above mentioned visual tools. As final step, the so generated rules are translated by the log management application into the native rule syntax, thus allowing the operator, to quickly setup a base set of rules, using simple and intuitive tools. This generated rule set can then be refined and optimized by adding or changing rules using the native rule syntax.

2.4 Summary

In this chapter, the architecture of current log management systems was introduced. After identifying input, processing and output layer as basic components of a log management system, requirements and best practices for implementing those basic building blocks were presented.

Since efficient handling and processing of large amounts of log messages represents one of the main requirements a log management system has to fulfill, special attention was given to log data analysis algorithms. After discussing advanced processing algorithms, such as for example event correlation, methods and existing applications for visualizing large amounts of log data were outlined.

3 Problems and Conceptual Issues with Current Systems

Although logging is nowadays implemented as core service in many operating systems or applications, the management and processing of the generated data is typically implemented by specialized applications. While most applications are only designed to filter the logged data using user-defined filters, thus being little complex and more viewer-type of applications than full fledged log management systems, last mentioned applications often face serious issues regarding interoperability and security or reliability. Therefore, in the following chapter requirements for modern log management systems are outlined and shortcomings of current systems are discussed.

3.1 Interoperability

On the following chapters I would like to discuss problems emerging from incompatibility issues faced when trying to integrate various logging systems into a centralized log management architecture.

Especially software systems built upon distributed systems or legacy integration frameworks are faced with the problem of data or protocol incompatibility. Furthermore, different system architectures form a further level of incompatibility. A log management infrastructure should therefore being easily adaptable to the large number of different logging implementations, since the goal is to obtain a comprehensive view over all concerned systems and being able to track error, status or other runtime events in a centralized manner.

Furthermore, also in terms of security, interoperability is a key requirement. For example, a domain-wide log policy covering security related events such as user access tracking or auditing, typically requires interoperability among the concerned systems, since the enforcement and monitoring of a security policy requires uniform access to the affected systems.

To achieve this level of system integration it is advisable to identify possible incompatibilities at the following processing levels:

- interoperability at data layer
- interoperability at transmission layer
- interoperability at storage layer

3.1.1 Interoperability at Data Layer

The most evident requirement for a centralized log management system is the ability to process every established log data format encountered on the concerned systems.

However, there is no commonly accepted log data format. In fact, as mentioned in chapter 1.2, a log is defined as a *record about an occurred event* and besides the timestamp no other elements are specified in the definition. This obviously permissive definition has led to a vast amount of different log message formats, ranging from simple formats, supporting only basic, timestamped status messages to more comprehensive ones, containing data fields for stack traces or other detailed debug information.

This rank growth of different log formats effectively circumvents building centralized logging systems and complicates defining a log policy for systems consisting of inhomogeneous logging facilities. Incompatibility at data layer also seriously affects the design of intrusion detection systems or log analysis applications, typically dependent on data collected by the logging subsystem. Those systems need uniform access to the logged data, since typically matching and event-correlation algorithms are used to reduce the amount of redundant information or, in the case of intrusion detection systems, trigger actions based on certain logging data event sequences.

Furthermore, certain parties are obliged to store their log data using standardized procedures and formats to comply with Federal legislation and regulations such as FISMA [KC93] or the Directive of the European Parliament on data retention [eu-05].

One of those application specific formats is the “W3C Extended Log File Format” [HBB96], an extensible format for logging statistical and diagnostic data generated by a Web server. It was designed as successor to the “Common Log File Format” [Luo95], used by one of the first available Web servers. Today, nearly every Web server supports logging to this log file format and many tools have been created in order to analyze and process logging data of those kind.

In the last years however, a standard for modeling semi-structured data, namely XML, has gained lots of attention, and there are ongoing efforts to convert proprietary logging formats to XML. In fact, the forthcoming Microsoft “Windows Vista” operating system uses XML as data format for log messages generated by the built in logging framework called “Windows Event Log” [vis06]. Another, well established logging framework using (among others) XML formatted messages is the Apache Logging Suite [apa06]. Other formats as LOGML [Pun01] or XLF [Par01] were announced and developed a few years ago, but have not gained much attention. Much of them are not being actively used or maintained.

However, the primary reason to favor XML over other, competing formats is the availability of a well tested and therefore mature tool chain. Basic procedures such as converting, processing and storing XML-based data are well established and supported by nearly every software system and programming language, so that using XML as log data format results in better interoperability with at the same time lower development costs. The goal of a log management system therefore is, amongst others, to provide means to convert the different formats into a common internal representation allowing to process messages from inhomogeneous sources in a standardized way. Typically this step is accomplished using one of the following methods:

- conversion at the source
- conversion at the processing host

The first method, conversion of native log messages to the common format at the source, is typically employed when the log management system is designed to process messages exclusively in the common format. Such architecture therefore implies that the message is either converted by the log-generating source, or by using conversion applications, often referred as *agents* or *proxies*, which convert the native format generated by the respective system into the format understandable by the log management system. This approach has the advantage, that the burden of the conversion process is imposed on the agent or log-generating application, which conserves processing resources on the log management system. The downside however is that either applications move out of the scope of the main application, the log management system itself. Getting their status or manage the configuration of the conversion process is therefore much more complex than conversion at the processing system. Furthermore, deploying the proxy application to a mission critical system or even adapt the application to the common log format is not always appropriate or possible due to the fact that each change to the currently running system typically affects its behavior.

From this point of view, the method of log message conversion at the log management system itself seems to be the most reasonable option. But like previously mentioned, implementing this additional feature directly in the processing core adds more complexity, thus error-proneness and processing power demands to the core application. Therefore, a compromise could be implementing this conversion process as plug-in, dynamically loadable at runtime. This alternative allows separating the conversion logic from the core logic, and at the same time provides convenient access to the conversion process configuration and data, since plug-ins typically run in the same address space as the core application.

3.1.2 Interoperability at Transmission Layer

Beside the format of the log message itself, also the transmission protocol used to transfer messages from the log-generating applications to the log management system needs to be considered. Protocols like the syslog protocol [Lon01], or the Windows EventLog remote access protocol [win06] are among the most used ones, but as with the log data format, there exists no standardized protocol to deliver log messages in a platform dependent way.

At first glance, the interoperability at transmission layer seems to be little important for building distributed systems, such as proxies or other protocol conversion applications provide facilities to interconnect rather incompatible systems. But having a single, well established protocol provides many advantages: Security reviews, for example, are performed much more easier on a single protocol rather than on a bunch of incompatible formats, not to mention the problems arising from side effects of using such configurations.

Furthermore, protocol-translating proxies may form a severe single point of failure in terms of scalability. While obtaining more capacity by adding more network resources to a single-protocol infrastructure is relatively easy, it might be difficult to scale systems using the previously mentioned protocol translators, since those systems might need to be adapted to the new conditions, instead of just scaling the amount of them.

Interoperability at protocol layer also simplifies the design process of distributed log management systems as such. The ability to choose from a range of protocol compatible components often assures that only the best choice for a given requirement can be used to implement the designed system. This approach avoids so called “work-around” solutions often needed to interconnect incompatible components.

3.1.3 Interoperability at Storage Layer

Another source of incompatibility arises from the format of the stored log messages. Log analysis tools are typically designed to gather the required data from the storage subsystem of a log management system, instead of being directly integrated into the system core. Such design is often preferred in order to provide an additional level of security. However, to be able to extract the needed information out of the logged data, the storage format of the latter must be compatible. Unfortunately, there is no common, standardized format available, apart from application-specific log format standards, such for example an approach for a log storage standard for digital libraries [GLS⁺02] or others. This circumstance is admittedly easily understandable, such almost every application needs to log different kinds and amounts of data, thus render the design of an all embracing storage format cumbersome. Moreover, considering that the format of the log message itself is hardly standardized, storing such data in an uniform manner is not an easy task.

Furthermore, almost every larger log system uses database back-ends for storing logged events. Subsequently, logging data is accessible only using the appropriate database drivers instead of direct access to the data pool. This fact is especially to consider when migration from inhomogeneous storage formats to a centralized storage pool is desired. In this case, typically conversion programs or proxy applications are used in order to convert the existent data into the new format. However, this scenario illustrates clearly the need for a standardized log format, since converting huge, possible terabyte-sized log repositories to a new format is not always possible, hardly ever in a decent time-span.

3.2 Dependability

Availability and reliability are probably amongst the the most wished properties of a modern computer system. In fact, there is little more frustrating than being dependent on a inherently unreliable system. By adding security and safety to those wishes one basically defines what in information science is called *dependability*. A dependable system is therefore a systems which is safe, secure and continuously available.

Since nowadays a software or hardware system is typically built upon a number of components, each component has to include means for achieving dependability in order to result in a dependable system as a whole. Therefore, also dependable log management systems or a logging subsystems as such have to be designed with that design approach in mind. In this chapter challenges and possible solutions for maintaining dependability will be outlined and compared with current implementations.

Before digging into details it is important to define what dependability is. The formal notion of dependability is generally defined as follows:

“The trustworthiness of a computing system which allows reliance to be justifiably placed on the service it delivers.” [dep06]

Dependability is typically further characterized in terms of the following attributes:

- Security
- Reliability
- Availability
- Safety

3.2.1 Security

As log messages may contain detailed information about the current system state, authentication events or other security sensible information, it is clear that each subsystem of a log management system needs to be protected against unauthorized access, in order to guarantee integrity and confidentiality of the processed data. In the following chapters, different security threats should be discussed and countermeasures should be outlined¹.

Confidentiality

Confidentiality is defined by the International Organization for Standardization as:

“Ensuring that information is accessible only to those authorized to have access.” [ISO05]

In terms of logging, confidentiality means that access to the logged data may be granted only to authorized parties. Typically, logging systems rely on the security infrastructure of the operating system, or storage subsystem in order to secure the access to the logged data. Syslog [Ger05], for example, contains no built-in feature to restrict access to the logged data, which is typically stored in a file. However, using Unix file system permissions to control the access to the logged data it is possible to deny unauthorized data access.

Limiting access to the logged data must also include the logging facility and it is configuration itself. Therefore, administrative access to the whole logging system must be secured, including configuration files or administrative consoles and all corresponding files of the concerned applications included in a logging system.

For example, the configuration file used to setup the syslog daemon must not be writable by non-administrative users or processes started by them.

Just like the stored data, also data transmitted to other systems via network links or other inter-process communications needs to be secured against unauthorized

¹for definitions and explanation of terms used in context with information security, please refer to [Shi00].

access. Typically such requirement is implemented by encrypting all related data before transmission. However, many logging systems are not equipped with such facilities. For example, the original syslog daemon uses unencrypted UDP packets to send log data across network links. Therefore, the transmitted data can be easily sniffed using appropriate applications. Using encrypted networks to tunnel such unsecured data, is therefore an absolute need when designing a log management framework which needs to be connected to legacy applications or networks using unencrypted data transfers.

Furthermore, no authentication or error checking is implemented in syslog, making it thus possible to intercept the data, modify it, and forward it to the actual recipient. In this case, neither the sender nor the recipient are able to notice this type of attack. To avoid such security threats, the syslog protocol was adapted to include authentication, message integrity, replay resistance and other security enhancing techniques as outlined in [CKC05].

Integrity

Just like confidentiality, integrity forms a basic requirement for a secure log management system. Typically, integrity is defined as:

“Safeguarding the accuracy and completeness of information and processing methods.” [ISO05]

Basically, this requirement implies that the system employs special precautions to ensure the processed and transmitted data cannot be altered or replayed, lost or otherwise misused. Such requirements are generally ensured by using checksums or other error detection algorithms, in order to guarantee the transmitted data was received unaltered. Typically integrity demands also authentication of the concerned parties, in order to avoid so called “man-in-the-middle” attacks, which may alter the transmitted data.

The original syslog implementation unfortunately features non of those requirements, since for example, peers involved in a transmission of log messages over the network are not authenticated, and the transmission is not secured at protocol layer. This is mainly caused due to the use of UDP as transport protocol without any other techniques used to avoid the mentioned types of security threats. Using syslog, the transmitted data is not secured against malicious or accidental alternation at protocol layer, using for examples checksums. As mentioned in the previous chapters, more recent implementations addressed those shortcomings by implementing RFC 3951 [NR01] or the proposal described in [CKC05] which recommends signed log messages for transmission over network links. Furthermore, many newer syslog implementations use TCP as transport protocol, additionally wrapped in encrypted TLS connections as defined in [DR06], respectively [BWNH⁺06].

3.2.2 Reliability

Typically, log management systems are designed to record data over a long term. In fact, most systems are intended to record data in an continuous way, in order to provide a complete view over the given time. Especially security related logs or log

data needed for statistical purposes, such as access or transfer logs need gap-less recordings of the configured events.

In order to fulfill this requirement, a log management system has to be reliable, which is typically defined as:

“The probability that a system will perform its intended function during a specified period of time under stated conditions.” [iee96]

This aspect has not to be confused with the term *availability*, which defines that a given system has to be available at the time of usage. Reliability however, refers to the *continuity* of a given system or service, as outlined in [AA01].

This requirement typically implies that the concerned systems and components have to be designed with *fault-tolerance* in mind. Fault tolerance denotes that a system needs to be operational also in aggravated conditions, e.g. on failure of one or more components.

However, in reality log management systems are faced with numerous threats which interfere with systems reliability. In the following chapter I would like to discuss the various categories of such threats, analyze how they are handled by current systems and propose feasible alternatives where appropriate.

Types of Reliability Threats

system overload: This reliability threat is one of the most often faced. Typically, software systems are designed to refuse further input in case of an overload situation. This approach allows the system to process the queued work in order to return to a normal operational state. However, to alleviate such excess load situations, especially systems receiving input data from other systems are typically equipped with a queue or buffer stage, in order to store received data instead of directly processing it. Therefore, in case the input data gets received at a larger rate than the system can handle, the buffer stage is used as temporary storage, thus avoiding load peaks.

However, simple buffering of incoming data to avoid system overload has a substantial drawback: As the system gets overloaded, and clients keep sending data at a high rate, the buffer stage keeps filling, and - after some time - reaching an upper limit, thus effectively detain any further reception of input data. The reason for this behavior lies in the fact, that the involved client systems are usually not aware of the overload situation at the server, since many transmission protocols are not equipped with an status “back-channel” in order to inform the client about possible errors or overload situations.

To overcome this limitations, transmission protocols such as TCP/IP [KC81] use various mechanisms, such as “sliding windows” [APS99] to avoid congestion situations. Therefore protocols for transferring log data should include similar congestion avoidance techniques or being designed to use appropriate transport protocols such as the mentioned TCP/IP. For example, early syslog implementations [Ger05] use UDP as transport protocol, which was explicitly designed to have none of the even mentioned features. Therefore, messages sent by the syslog daemon to a remote receiver are transmitted without any

guarantee, since there is no reception acknowledgment or error recovery implemented in the syslog protocol [Eat03].

It is important to note that such overload situations are not necessarily generated by authorized client systems only, but, especially when a system is directly connected to “inhospitable” public network such as the Internet, also by malicious ones. In fact, in the last years an increase of so called denial-of-service attacks has been observed, so nowadays these type of attack have to be considered when designing reliable systems.

In certain situations, however system malfunction may persist for longer time spans, thus over-straining even the most complex buffering and anti-overload algorithms. In this case, depending on the importance of the affected system, various possible reactions are feasible:

continue in degraded mode: If the logged data is not mainly used for debugging or other non critical purposes, such as statistical objectives, the system may proceed in a so called degraded mode, with disabled logging, typically falling back to normal operation mode after the overload situation is averted.

perform evasive actions: In this case, the application tries to perform predefined actions in order to avoid further aggravation of the system state. A log management system, in case of low storage, might initiate (after issuing an appropriate administrative alert) preventative log rotation, in order to rotate and compress old log data and gain additional storage space.

emergency shutdown: The last possible way to safely react on overload situations is to cleanly shutdown the overloaded system, in order to guarantee proper termination of open file handles or other runtime resources. It is important to note that such behavior may lead to high sensibility to denial-of-service type of attacks, thus possibly threaten not only the logging system itself, but other log dependent system components.

network unreliability: Just like system overload, network unreliability forms a serious threat to system’s overall reliability. Since log management systems typically collect log messages over network links, the reliability of this transfer medium and consequences for the transfer of log messages on network failures must be considered. A key requirement in this case is that transferred log messages must not get lost, replayed or otherwise modified under any circumstance. This requirement implies that the transport protocol must guarantee reliable, in-order delivery of log messages from the sender to the destination. TCP/IP [KC81], for example, is such a protocol, thus being qualified as low-level transport protocol for log messages. In fact, almost every modern log transfer protocol, such as RPC/DCOM on the Windows Eventlog [win06] or recent syslog implementations [NR01] use this protocol as reliable foundation for transferring log messages to remote systems.

However, it is important to address the problem of unreliable transmission of log messages also at application layer, rather than exclusively rely on the

underlying transport protocol. Reliable logging systems respectively protocols therefore use acknowledging algorithms also at application layer, in order to guarantee an error-free transmission.

3.2.3 Availability

Besides the security threats a log management system may be faced with, another important issue that needs to be considered while designing such a system is availability. Availability is typically defined as:

“Ensuring that authorized users have access to information and associated assets when required.” [TS02]

This definition implies that a system should not expose any service interruption or degradation in case of an error or overload situation. This requirement is especially important for logging or log management systems as the those subsystems typically represent the only source of detailed information about an error condition or other diagnostic data in case of a system failure or interruption.

Therefore, special precautions need to be considered, in order to render the log system highly available. In the following chapter, I would like to outline specific threats regarding log system availability and discuss counteractive measures in order to circumvent such conditions.

Clustering Architectures for Logging Systems

As availability demands form a large portion of the overall requirements a log management system must fulfill, larger system are typically realized using clustering architectures.

A cluster, in terms of computer systems, is typically a compound of loosely or closely coupled machines, which typically form a distributed system. In fact, Tanenbaum in [TS02] denotes a *distributed system* as

“Collection of independent computers that appears to its users as a single coherent system.”

In a log management system, different sub-systems such as the input layer and the storage layer may be implemented using a clustered architecture, in order to increment performance and reliability. Typically, such systems use one or a combination of the following cluster types:

high-availability clusters: High-availability clusters are used to increase the availability of a system component or system as a whole. In a log management such systems are typically employed to minimize the shortcomings outlined in chapter 3.2.3.

load balancing clusters: This category of clusters is usually implemented in order to spread the load of a single system, the so called *front-end system to back-end systems*. In addition to a increase of performance also the reliability of a system typically increases, provided the front-end system provide means to guarantee availability.

From this basic cluster types it is possible to build customized architectures, as for example in the following picture 3.1.

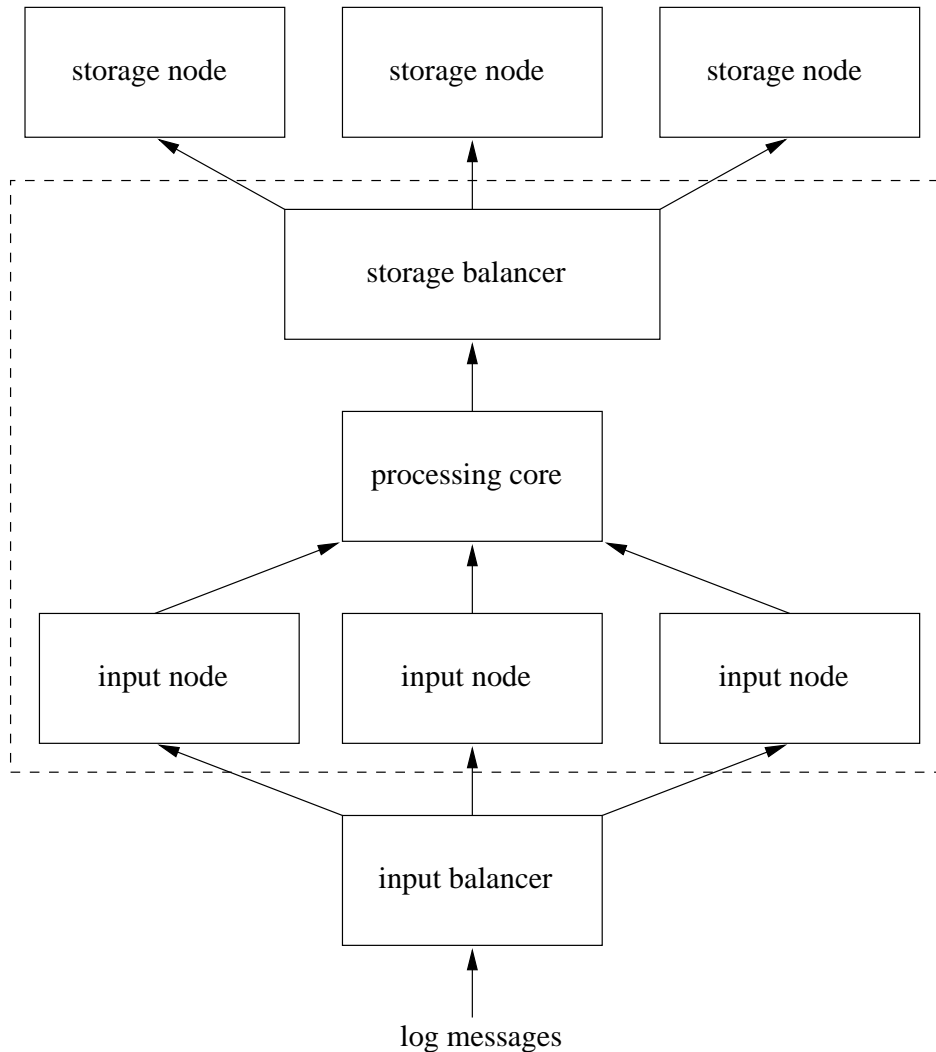


Figure 3.1: Cluster architecture example

In this example, the input layer of a conventional logging system (outlined in figure 2.1) is replaced by a load balancer system, redirecting the incoming stream of log messages to a cluster of processing systems, that forward, for their part, the processed result to another load balancer, distributing the results to a clustered storage back-end. Such storage back-end might be a clustered file system, providing space for log data storage, or a cluster capable database such as for example used in the commercial log management system SenSage SAS [sen06].

Depending on the actual requirements also other cluster architectures for log management systems are feasible. For example, the above mentioned model has two “bottlenecks” or rather “single point of failure”, namely the two load balancers for the input and output system. In the case of an error or fault of one of those

subsystems, the whole infrastructure is affected, because either the input or the output system is effectively inaccessible if one of both subsystems fails. It is advisable therefore to absolutely resign such functional dependencies in order to guarantee a certain level of reliability. A best practice for avoiding such design pitfalls is to check each subsystem or component of a systems architecture regarding availability issues.

Simpler architectures do not dispose dedicated load balancers in order to redirect the incoming traffic to so called “back-end systems” but rely on the client to carry out simple load balancing algorithms. For example, the syslog protocol [Lon01] supports sending log messages to other systems. Together with the possibility to configure multiple destination hosts this forms a basic form of load balancing, since messages are sent simultaneously to each of the configured remote servers. Since the syslog protocol is UDP-based and sending UDP packets to non reachable destinations (unlike TCP) does not impose any timeouts or consumption of system resources other than those of sending the packet, this kind of simple clustering is an alternative to full blown clustering systems.

To achieve the previously mentioned requirements regarding systems availability and in general systems dependability, fault tolerance plays an important role. A fault-tolerant system is generally a system that is able to operate even on a failure of one or more of its components. In order to guarantee a proper operation even in aggravated conditions, not only the system software and hardware has to be designed with that design goal in mind, but also the inter-process communication facilities, such as for example network connections have to be included in the design process of the fault-tolerant system.

3.2.4 Safety

Safety is typically defined as

“Absence of catastrophic consequences on the user(s) and the environment.” [AA01]

In the domain of log data processing and log data management this requirement is at the first sight implicitly satisfied, as typically log management frameworks are mainly passive system components, not interacting with the surrounding system environment. However, as the log management system accesses system resources like storage subsystems or databases, this way unintentional interferences with other system components may arise. For example, extensive logging activity might consume considerable amounts of processing power or storage resources, and therefore encumber other important systems, possibly leading to system malfunction. Therefore, a log management system should be designed to use only defined amounts of the available system resources in order to guarantee reliable operation even on severe conditions. Also the coupling with other system components should be designed as loose as possible, therefore avoiding dependencies which typically lead to complex and unpredictable system behavior in error situations.

It is important to note that exactly logging subsystems often provide the only valuable input for debugging and forensic purposes in case of a system failure. In order to not falsify the logged data by interferences caused by the logging system

itself, the logging system should be therefore designed as robust and unobtrusive as possible.

3.3 Portability

In addition to the above mentioned requirements for a log management system, also portability is a desired feature of a modern log management system.

Portability is typically defined as:

“An application is portable across a class of environments to the degree that the effort required to transport and adapt it to a new environment in the class is less than the effort of redevelopment.” [Moo97]

Since larger log management systems are usually implemented using distributed components, either for achieving high availability as described in chapter 3.2.3 or to improve scalability, the possibility to spread those components on actually incompatible systems is a great benefit. Implementing those modules using a portable programming language or runtime environment facilitates the migration of the entire system or specific modules to powerful machines, thus generally improving systems overall performance.

An example of such a system is the Apache Logging framework “log4j” [log06], which is built on top of the Java virtual machine. This portable runtime environment allows straightforward migration of the logging framework to a large number of machine architectures without recompilation or reconfiguration. For further description of the log4j framework, please refer to chapter 4.2.

3.4 Scalability Issues

Looking at the massive gain of performance of computer systems in the last years and defining logging as one of the most important building blocks of modern software or hardware systems it is clear that the architecture and implementation of log management systems needs to scale in order to guarantee reliable and timely processing of log messages also in demanding environments. The term *scalability* is typically used to refer to a systems availability to meet elevated workloads.

More specifically, scalability is defined by Bondi as follows:

“The ability of a computer application or product (hardware or software) to function well as it (or its context) is changed in size or volume in order to meet a user need.” [Bon00]

In addition to the problems concerning availability, and cluster architectures for log management systems discussed in chapter 3.2.3, scaling a such systems in order to process elevated workloads poses several issues, which can be categorized into the following categories [Bon00]:

- load scalability
- space scalability

- space-time scalability
- structural scalability

In the following chapters the different types of scalability will be outlined and problems of log management regarding the respective type are discussed. After comparing current implementations in respect to their scalability, possible solutions for improving systems availability will be proposed.

3.4.1 Load Scalability

Load scalability is probably the most often discussed scalability type. In short it denotes the capability of a system to function gracefully under each load [Bon00]. For example, the Ethernet protocol using CSMA/CD is not load scalable, since its performance declines heavily in overload situations, in extreme situations even leading to saturation. For a log management system such behavior is definitely unacceptable, since this specific systems are essential for recording overload situations for later analysis.

Before identifying load scalability issues in the specific subsystems, two main principles for addressing load scalability issues, namely *loose coupling* [Wei01] and *pipelining* should be mentioned:

By loosely coupling systems components, the effect of inter module dependencies concerning load can be minimized. In log management systems, the basic components such as the input, processing and output system should be connected without introducing dependencies. Overload situations striking a single module should not affect the other ones. Ensuring this basic requirement simplifies scaling each module independently and as result scaling the log management system as a whole.

To avoid load scalability issues, large log management systems are typically based on cluster architectures. Especially computing intensive log analysis tasks are often deployed on multiple machines, in order to distribute the workload. However, in order to achieve uniform utilization of all resources in a cluster system, the underlying algorithms of log message processing need to be scalable.

For example, scaling pattern recognition and filter algorithms is typically accomplished by subdividing the input data into multiple parts, and assign each part to a specific machine in a distributed system. However, special precautions have to be considered, in order to avoid that multiple processes read from the same range of input data, which often leads to overlapping results. Especially data extraction or reduction algorithms, which are typically used to condense multiple events into one, representative event, may produce multiple identical and thus misleading results if the range of input data overlaps.

As mentioned, pipelining is another method for improving a systems load scalability. Typically pipelining is used when the same data passes through several processing steps, which are consecutively dependent on each other.

3.4.2 Space Scalability

Space scalability is typically defined as:

“A system or application is regarded as having space scalability if its memory requirements do not grow to intolerable levels as the number of items it supports increases.” [Bon00]

On log management systems this requirement affects typically the processing subsystem, which processes all incoming messages and decides its further path across the system. In order to fulfill the requirement for space scalability, the processing core has to be designed with memory limits in mind. Typically such requirements are implemented using algorithms with predictable memory consumption or by limiting the maximal amount of memory per process to a reasonable number. The latter approach has to be especially considered, if the processing core uses multiple processes in order to analyze the received data, as typically the amount of needed resources increases linearly with the amount of participating processes.

3.4.3 Space-Time Scalability

Weick defines space-time scalability as

“We regard a system as having space-time scalability if it continues to function gracefully as the number of objects it encompasses increases by orders of magnitude.” [Bon00]

As log management systems may use elaborate algorithms for processing and filtering incoming log events, in order to guarantee space-time scalability, the choice of algorithms should prefer algorithms, that are

“conducive to smooth and speedy operation whether the system is of moderate size or large.” [Bon00]

Basically, this requirement demands that the algorithms runtime and space consumption ranges between linear and logarithmic values. As the processing core of a log management system typically uses search and string matching functions, it is advisable therefore to avoid algorithms with worst case $O(n)$ runtime and space consumptions.

3.4.4 Structural Scalability

Structural scalability basically demands that a system should not impose any limits in its data structures or operating environment (as described in [Bon00]). For example, a storage subsystem, allowing only a certain amount of datasets to be indexed conflicts with the definition of a structural scalable system.

To achieve full structural scalability, also the interprocess or network protocols used to build a log management system need to be considered. For example, if the network protocol is limited in its addressing scheme it compromises system scalability.

However, current software and especially hardware often enforces certain structural limits, be it because of technical limitations or deliberate defined limitations. Therefore, achieving full structural scalability is not easily achievable using nowadays computing systems.

3.5 Architectural Issues

Besides the mentioned issues and problems concerning the implementation of a log management systems also architectural issues need to be considered during the design of a such system.

A large amount of currently available log management systems tend to integrate all possible functionality in one single application or system. While such approach typically leads to software systems with impressive functional range, it often violates an important paradigm in modern software development, denoted to as “the separation of concerns” [Dij03]. The separation of concerns proposes that each functional subsystem or module of a system should be cleanly separated from other system components and encapsulate exclusively functionality that is absolutely required for the specific task. Typical implementations of this paradigm can be found in software patterns like the “model-view-controller” pattern as well as in recently often discussed service-oriented architectures.

In the field of log management systems this paradigm may be implemented by strictly modularizing the various system components in logically separated and self-contained modules. Each module should only contain the business logic needed to accomplish a clearly defined task as well as a precise interface in order to communicate with other components.

On a higher level of abstraction, this requirement might be fulfilled by structuring a log management system into the mentioned basic building blocks, namely input, processing and output layer. At more lower level, the individual layers should be further designed with the separation of concern paradigm in mind. For example, the output layer should not be realized in a monolithic way, implementing each eligible feature, but separated in smaller, lightweight modules. In the case of the output layer, a possible approach might be to implement each protocol the output layer should support as separate plug-in.

This architectural decision allows flexible configuration and wiring of the concerned modules and therefore easy adaption of the system to changing requirements. Furthermore, an often neglected aspect, namely testing and formal verification of the whole system is greatly facilitated by separating the concerns, since each module may be individually tested. Since the codebase for each module is typically of moderate and thus manageable size, especially the formal verification of a module is therefore greatly simplified - which generally leads to reliable implementations.

3.6 Summary

Since modern log management systems are typically complex distributed systems and in many cases the only source of error details and debugging information such systems need to be designed for reliable operation even in heavy load situations.

This chapter outlined problems and architectural issues which may arise during the design or operation of log management systems. After classifying them into four main categories, namely interoperability, dependability, portability and scalability issues, possible solutions, architectural patterns and best practices in order to address such challenges were presented.

3 Problems and Conceptual Issues with Current Systems

4 The XMPP Log Management Framework

In the previous chapters different approaches for designing a log management system were discussed. In this chapter, an alternative approach of such a system will be outlined, and advantages over existing implementations will be presented as well as possible drawbacks.

4.1 Overview

The basic idea for the XMPP Log Management Framework originated while evaluating log4j [log06], a well known and established logging implementation for the Java platform. Since log4j is actively developed and released under the Apache License Version 2, many programmers have contributed code in order to extend log4j's capabilities. Therefore, with the forthcoming log4j version 1.3 many unique features were merged into the codebase, making it the logging system of choice for the Java platform and beginning with the latest version, also for other system platforms.

However, log4j was originally intended as simple to use and configurable logging framework. In fact, the wish for a Java-based logging framework, providing an uniform and configurable interface for all types of logging purposes was often stated, since many applications at that time wrote their logging output simply to the terminal or to files. The code for such log statements was typically constructed by simple "print" commands, combined with simple constructs such as

```
if (LoggingEnabled()) {  
    System.err.println("connection established");  
}
```

for conditional logging. Log4j obsoleted such rather rudimentary and inflexible logging approaches by defining a framework consisting of two basic components:

- a well designed programming interface
- a structured and adaptable configuration system for end-users

The programming interface allows the developers to concentrate on the application logic and delegate all logging related low-level details to the log4j library. For the end-user on the other hand, log4j provides easy and consistent configuration through simple property-style or XML-formatted configuration files. Furthermore, the log4j API permits configuration at runtime, where the configuration is completely separated from the actual programming interface, thus supporting extremely

customized configuration scenarios. This design approach, together with a specifically speed optimized logging core makes log4j also appropriate for demanding logging purposes.

While many Java developers choose log4j as logging framework, others begun providing mechanisms and applications for visualizing the logged data. For example, projects such as LogFactor5¹ by Servidium Inc. (now ThoughtWorks Inc.) and Chainsaw, written by Oliver Burn [Bur02] lead to the development of Chainsaw V2², the log-viewer application shipped since log4j version 1.2. Chainsaw was designed as log4j integrated graphical viewer for logged events, allowing the user to sort, filter and export log events generated by the log4j subsystem. Furthermore, the system architecture is based on a heavily modularized design, making it possible to extend the functionality in various ways. This fact, indeed was one of the main reasons to use log4j, respectively Chainsaw as foundation for the proposed prototype.

4.1.1 Features

The XMPP Log Management Framework is primary intended as prototype in order to provide a test bed for a new log management application design using the XMPP messaging protocol [SA04b] as transport protocol and the Chainsaw log viewer as basis for the user interface. Therefore, all messages generated and received by the logging framework (for detailed information about the system design, please refer to chapter 4.2) are encapsulated and transmitted as XMPP messages. This, at the first sight unusual approach, to use a protocol originally designed as transfer protocol for instant messages as transport protocol for a log management application offers many advantages:

Reliable Infrastructure: Since reliable transport of log messages from the sender to one or multiple destinations is one of the key requirements for a logging system, the adoption of a well established and reliable protocol such as XMPP as transport protocol is a promising choice. By using TCP as underlying transport protocol, and XML as format for the transported messages, two well known and mature technologies build the foundation for XMPP and therefore for the proposed logging framework. Unlike other logging implementations, which use unreliable UDP as transport protocol, TCP allows reliable transmission of data over unreliable network links, which is especially needed for important information such as log messages are.

Integrated Security: XMPP has builtin support for SSL or TLS encryption [SA04a] for the entire transport data stream and supports certificate-based end-to-end signing and payload encryption using the SMIME [Ram04] format. Peer

¹LogFactor5 was entirely donated to the Apache Software Foundation in April 2002 [Mar02] in order to be integrated in log4j. Since the development of LogFactor5 stopped at this event, there is no documentation or website available for further reference.

²since the original Chainsaw application is not being developed anymore and all of its functionality was merged into Chainsaw V2, in the following chapters, the term “Chainsaw” is used instead of “Chainsaw V2” for easier reading.

authentication can be carried out using algorithms supported by the SASL [Mye97] library, such as DIGEST-MD5 and CRAM-MD5.

Interoperability and Platform Independency: XMPP is an open protocol, standardized by IETF in RFC 3920 [SA04b] and RFC 3921 [SA04c], therefore all needed details for programming software using the XMPP protocol is available freely. The main development is organized by the Jabber Software Foundation [jsf06], a non profit organization. Furthermore, all protocol extensions, such as for example security or feature extensions typically use well established standards.

At the time of writing the Jabber Software Foundation lists more than 10 freely or commercially available server and more than 30 XMPP client implementations. Furthermore, libraries for nearly every programming language and operating system are available, making development of applications for XMPP easy. As at these days, mobile communication plays an important role, it is important that a new client technology can be used also by mobile systems. In fact, there exist many implementations of XMPP clients for mobile devices such as organizers and PDAs.

Just as the choice of TCP as underlying transport protocol for reliability reasons, it guarantees also interoperability amongst a broad range of operating systems and well established network infrastructures.

Extensibility: As XMPP uses XML as data format, extensions are easily implementable. New extension proposals can be submitted to the Jabber Software Foundation, where they are approved and voted by the XMPP Council and in the positive case proposed as XMPP Extension Protocols. Currently there exist more than 30 proposed extensions, ranging from integration of service discovery to advanced messaging proposals including time-sensitive messages and bindings for encapsulating XMPP messages in HTTP requests.

Widespread Usage: A not so unimportant factor for the success of a new application or technology is the willingness for adoption by the intended user. Applications built upon proprietary or less known standards are typically avoided by end-users. A main advantage for using a instant-messaging protocol for a log management system, is the large degree of popularity of such protocols. Nearly every Internet user has at least some experience in using an instant messenger. By integrating an instant messenger protocol as basis for a log management system, the learning curve to use such a system can be significantly lowered, thus making the system more easily accessible by the end-user.

The above listed features of XMPP make it possible to focus on the application logic of the log management system, letting XMPP and the underlying TCP protocol handle the details of reliably and securely transferring the log messages throughout the systems. Moreover, it is easy to implement alert and notification mechanisms through instant messaging, since instant messengers are typically built in order to allow quick and responsive messaging across a large number of interconnected systems.

Furthermore, the goal was not to design a new log management system from scratch, but to research if the modular design of Chainsaw can be extended in order to integrate all main features of a log management system.

4.1.2 Back-Channel Feature

Since many of the existing log management systems are configurable only through configuration files or the graphical user interface, the XMPP Logging Framework should be extended in order to allow configuration of the system also from remote machines. This so called “back-channel”³ is especially useful, when certain situations demand immediate configuration changes. For example, overload situations or denial of service attacks might require immediate re-configuration of the logging framework in order to react on such exceptional events.

Furthermore, in certain use cases such as high security environments it is desirable to separate the access to systems configuration from the configuration of the logging framework. In this case, the back-channel feature might be used in order to allow remote configuration of the logging infrastructure, and delegate this access to people without other system access privileges. For example service providers, offering system monitoring services to other parties might be interested in such systems in order to clearly separate logging and configuration of the logging process from the other system services and without the need of real system accounts.

Since the core application uses XMPP, the idea was to use this protocol as well for administrative access. In fact, as mentioned in chapter 4.1.1, XMPP features integrated user authentication and secured data transfer using TLS or SMIME. This security features encouraged the choice of using XMPP as remote access protocol. Besides the strong security features, XMPP has also the advantage of being well supported by all recent instant messengers.

The idea is therefore, to expose the administrative interface through a XMPP chat service. The administrator may therefore connect to the administrative interface by initiating a chat with the given system. To issue commands, as for example to list all configured log sources, a simple chat-message, containing the appropriate command and optional parameters is sent to the server, which on his part replies with a message containing the result of the command.

Using this rather simple text-based protocol it is possible to use virtually any XMPP client in order to configure the proposed log management system and receive status or alert notifications. Especially resource-limited mobile devices, such as PDA's, or mobile phones can therefore be used for remote administration.

4.1.3 Improvements over Existing Systems

The main advantage of the XMPP Log Management Framework compared to other systems is its modularity. By using log4j as foundation, and building the log management system as plug-in to the existing log4j framework, the log management

³The feature was named “back-channel” considering that in typical logging applications, the flow of data is from the log-generating system to the log management or storage system and therefore no or little data is exchanged in the reverse direction.

system is based upon a highly modularized design. This feature allows easy adaptation of the system to various requirements. An example of a possible configuration is given in figure 4.1.

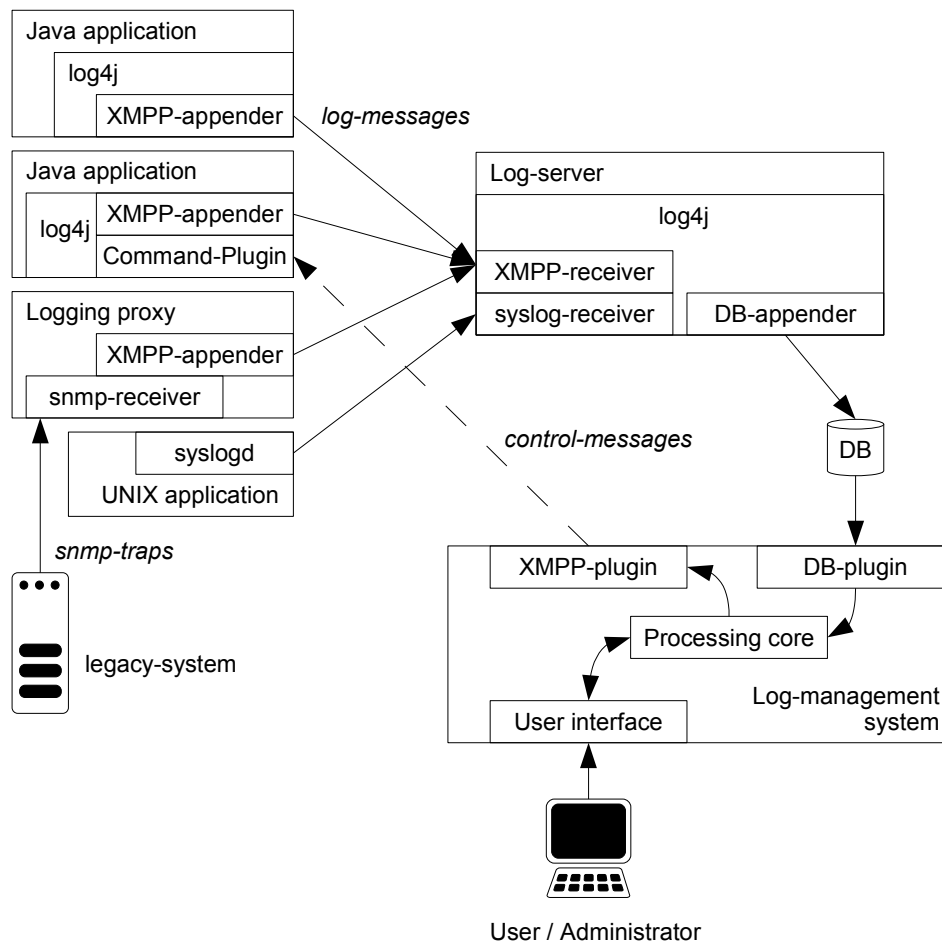


Figure 4.1: An example integration

The example configuration shows how the log management system may be integrated in a Java system, where log4j is used in as foundation. In order to integrate log messages from non Java-based systems, such as the SNMP capable legacy system a logging proxy is used in order to transform SNMP traps into log4j messages. For integrating syslog messages into the system, the log server log4j configuration was adapted to include a log4j syslog receiver component, which transparently imports syslog UDP datagrams into the logging server. The example further shows how the actual log management system, including components for processing log messages and providing the user interface may be logically decoupled from the receiver and appender subsystems. Such approach allows creating lightweight logging infrastructures, as only the appender respectively receiver components need to be deployed on the log sources. The administration and processing of the recorded data is therefore hosted on a dedicated machine, thus clearly supporting the “separation

of concerns”.

Another advantage of the XMPP Log Management Framework is its platform independency due the choice of the Java-based log4j framework respectively Chainsaw as foundation for the system. Furthermore, the system can be simply copied to the target system, as no installation is needed.

By further relying on XMPP for the transfer of log messages among the concerned systems, user authentication and optional encryption of all transferred data is built right into the base system. By using JAAS, the *Java Authentication and Authorization System* for authentication and authorization, the system is able to use virtually every authentication scheme, and can therefore take part of a single sign-on system. Furthermore, the choice of XMPP as transport protocol promises great interoperability and accessibility combined with reliable transfer of log messages across possible unreliable network links.

4.2 System Design

In this chapter, the technical architecture of the XMPP Log Management Framework will be outlined and possible extensions to the existing architecture will be presented.

Since the log management system uses log4j as foundation and also the graphical user interface, realized by a Chainsaw plug-in uses log4j, i would like to give a short overview of its technical architecture before introducing the architecture of the log management system:

Basically, log4j was created as logging framework for Java applications. It consists of a core library, implementing the core functionality and a number of optional extension libraries. Just like traditional printing of debugging messages to the standard output stream or to files, log4j provides an API in order to post a messages to the logging infrastructure.

In order to redirect the posted log messages to various destinations, including files, output streams or other destinations, log4j can be configured in manifold ways, where its configuration is strictly separated from the logging interface exposed to the application. This separation of the logging interface from its configuration is one of the main features of log4j. In order to understand the internal mechanisms used by log4j, it is however necessary to introduce the following basic components and concepts of log4j:

Level: In order to be able to separate the generated log messages into different, discrete categories, log4j supports the notion of a *level*. For example, a log message destined for logging purposes might categorized using the “DEBUG”-level, whereas an important log message generated on a system failure might get categorized using the “ERROR”-level. By introducing a hierarchy of level, it is possible to compare and sort logged messages by their “severity”, as outlined in chapter 1.2.1. Log4j predefines 6 levels, ranging from “TRACE” to “FATAL” [log06]. By evaluating the log level, log4j decides if a logging request by the application should be *enabled* using following rule:

A log request of level p in a logger with (either assigned or inherited, whichever is appropriate) level q , is enabled if $p \geq q$.

Logger: The concept of a *logger* is used by log4j in order to provide the possibility to separate a number of logging streams from each other and to allow building a hierarchy of loggers. At the root of the logger hierarchy a predefined *root logger* is automatically created by log4j. Applications can request new loggers from log4j, and form a hierarchy by using Java package-style Logger names. For example, the logger “org.trispace” is a child of the root logger, and “org.trispace.warning” is a child of “org.trispace”. After requesting a new logger from the system, they are ready to *generate* log messages by calling their the appropriate logging calls, such as “info()”, “alert()”, etc. . .

Forming a hierarchy has the advantage of being able to inherit properties from the top of the hierarchy to its leaves. Therefore, properties appropriate for all loggers may be defined at the root level, letting child loggers inherit this basic configuration, in order to provide more consistent configuration.

Appender: Appenders form the destination for log messages generated by one or more loggers. In order to support multiple destinations for a log message, log4j supports multiple appenders. Currently, log4j is shipped with appenders for files, sockets, JMS and other output streams.

The connection from a log-generating source, or more specifically, a logger to its destinations is configurable (even at runtime) by using one of log4j’s configuration schemes.

Layout: In order to be able to specify the format a log message is being written to the appender, log4j supports so called “layouts”. An example for a layout would be a HTMLLayout class, formatting a log4j log message as HTML-fragment.

Receiver: Beginning with version 1.3, the log4j framework was extended to provide so called receivers. Before this concept was introduced, the only way to generate log messages was through an API function call. With receivers, the framework is able to receive messages from arbitrary sources and processing the so received messages as if generated locally.

The concept of receivers therefore allows creating centralized logging servers, receiving log messages from external systems. Furthermore, the concept of receivers is not limited on network-type communications only, but it is perfectly feasible to use, for example a JDBCReceiver to extract log messages from a database in order to import log messages from a legacy or incompatible system into a log4j infrastructure.

An overview of the log4j framework is given in figure 4.2.

4.2.1 System Description

The XMPP Log Management Framework extends the log4j framework by adding the following new components:

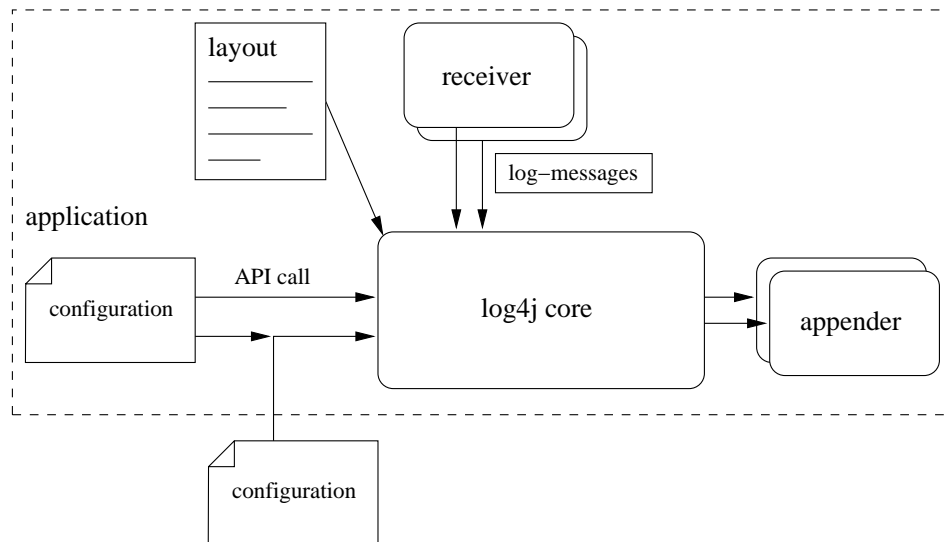


Figure 4.2: Log4j overview

XMPP-Receiver

The log4j receiver system is extended by a XMPP-Receiver, which allows reception of log messages using XMPP as transport protocol. In order to receive XMPP messages, the XMPP-Receiver needs to be logged on to a XMPP server. From the server point of view, the XMPP-Receiver appears just like a common XMPP user.

However, in order a received XMPP message with an encapsulated log4j-event gets recognized by the XMPP-Receiver, the XMPP message needs to be formatted by using a certain format. More specifically, after a XMPP message is received by the XMPP-Receiver its `body`-element gets decoded by a configured *decoder*, which generates a native log4j `LoggingEvent` out of the received XMPP message body. By default, an `org.apache.log4j.xml.XMLDecoder` is used, although it is possible to configure alternative decoders by subclassing the `org.apache.log4j.spi.Decoder` interface and assigning the class name of the new decoder class to the XMPP-Receiver. The class is then automatically loaded by the class loader right after the XMPP-Receiver is instantiated. This approach allows customizing the format of the XMPP message's payload, without changing the XMPP-Receiver code.

In order to connect to a XMPP server, the XMPP-Receiver needs to be configured with the hostname and port of the XMPP server and with the user-credentials of a existing XMPP account. Moreover, it is possible to setup whether a SSL-connection should be used or not.

XMPP-Appender

The XMPP-Appender is the counterpart of the XMPP-Receiver, since it is used to send log4j-events to a given XMPP contact. In order to send messages, the XMPP-Appender - just like the XMPP-Receiver, must be logged on to a server. Therefore, user credentials and the XMPP hostname have to be supplied before a connection

can be established.

Furthermore, the destination XMPP contact, called *receiver* must be set. Just as the XMPP-Receiver, the format for transforming a `log4j-LoggingEvent` can be configured. The XMPP-Appender, however uses a `Layout`⁴- class in order to convert the `LoggingEvent` that is to be sent, into a format, appropriate as payload for a XMPP message.

These two core components of the proposed logging framework, the XMPP-Appender and the XMPP-Receiver may be used also independently. For example, the XMPP-appender might be useful for inclusion in a existing log4j configuration in order to alert fatal or important log messages to a configured XMPP account.

XMPP-CommandPlugin

The XMPP-CommandPlugin implements the back-channel functionality, as discussed in chapter 4.1.2. As the name suggests, the XMPP-CommandPlugin is implemented as runtime loadable module and is typically configured in the respective log4j configuration file. A typical configuration of the command plug-in is given in the following example extract of a log4j-Joran⁵ configuration file:

```
<plugin name="XMPPCommandPlugin"
  class="org.trispace.xmlpp.XMPPCommandPlugin">
  <param name="host" value="trispace.org"/>
  <param name="username" value="mailserver"/>
  <param name="password" value="mailserver"/>
  <param name="useSSL" value="true"/>
  <param name="port" value="5223"/>
</plugin>
```

As seen in the example configuration, the XMPPCommandPlugin is configured just like any other XMPP client, in fact, the plug-in appears to the end-user as simple XMPP contact. In order to access the plug-in, the user opens a chat to the configured user name and may execute actions by sending an appropriate command line to the XMPPCommandPlugin.

In order to allow easy customization and ensure authorized access, the XMPP-CommandPlugin is subdivided in two major modules:

The Parser Repository The parser repository contains a list of mappings between a *command* and the corresponding *action*.

An *action* can be any class extending the abstract class `Action`, which defines an `execute` method and provides an overrideable property called *command*. This property defines which command line should be used to trigger the given action defined by overriding the `execute` method.

⁴unlike the XMPP-Receiver, which uses a `Decoder` class. Basically the `Decoder` class is the counterpart for the `Layout` class and vice versa.

⁵Joran is the name of the log4j XML parser used to process the various log4j configuration files.

Since the repository contains all available actions, it needs to be defined where to look for available actions. This configuration is handled by a so called `ConfigurationManager`, which is activated at system startup. In order to allow greatest flexibility, the `ConfigurationManager` evaluates a JVM runtime property defining which class should be used for configuring the repository. This approach allows adapting the configuration of the parser repository to the given requirements such as embedding the `XMPPCommandPlugin` in environments, where the system initialization and configuration is handled by a container, for example J2EE, respectively JSP containers or application servers.

Currently, only a simple, static configurator, the `SimpleConfigurator`, which configures the repository from using a hard-coded list is implemented. By implementing the `Configurator`-interface it is however possible to create customized configuration classes for the `XMPPCommandPlugin` like for example, configuration of the parser repository from a database or by using the Spring Framework.

The Security Manager As the back-channel feature allows extension by arbitrary action classes which are accessible by a instant messenger, it is often desired to restrict the access to this administration interface. For example, it is not advisable to let security sensible actions, such as changing the logging configuration be accessible to all users. Therefore, the XMPP Log Management Framework allows securing the `XMPPCommandPlugin` using the Java Authentication and Authorization Service (JAAS, [jaa06]).

The base component of the security infrastructure is the `Security Manager`, which has methods for logging in, respectively logging out a currently active user and performing a JAAS checked invocation of a given action.

The execution flow of a typical user interaction with the `XMPPCommandPlugin` is as follows:

1. The user starts a chat with the `XMPPCommandPlugin` and sends a command
2. Further access from other users is blocked and a “connection busy” message is displayed. This proceeding is needed to assure exclusive access to the administration interface.
3. The `XMPPCommandPlugin` parses the command, by consulting the parser repository, and retrieves the respective `Action`.
4. The `Action` is handed over to the `SecurityManager` in order to check the access for the given action and execute it.
5. The `SecurityManager` creates a new JAAS `SecurityContext` in order to log-on the user to the JAAS subsystem. To retrieve the user credentials, the JAAS system calls the configured `CallbackHandler`, which requests the user credentials and returns a `LoginContext`, which finally invokes the user log-on.⁶

⁶in order to test the `XMPPCommandPlugin` without the need of a XMPP connection, the JUnit tests use a `PassiveCallbackHandler`, which simply returns a previously configured user name.

In the default configuration, the credentials are retrieved by asking the XMPP subsystem for the currently logged in user. By configuring other `CallbackHandlers`, it is possible to retrieve the user credentials using arbitrary methods, such as requesting it via a GUI-Popup or gaining it from a chip card.

In order to check if the credentials are valid, the JAAS system asks the configured `LoginModule`. The configuration of this module is done within the scope of JAAS configuration, as outlined in the JAAS system documentation [jaa06]. In the current distribution, JAAS is configured to use a `UsernameLoginModule`. This module checks a given user name against a hard coded list of valid user names. In productive environments it is obviously wise to configure JAAS to use a more sophisticated `LoginModule`, such for example, a module authenticating against the system user database.

6. If the log-on procedure succeeds, the `SecurityManager` stores the user credentials, in order to skip the above mentioned steps after subsequent commands from the logged in user. The control is given back to the `XMPPCommand`.
7. The `XMPPCommand` hands the previously created `Action` to the `SecurityManager` in order to check the access against the now logged in user.
8. The `SecurityManager` wraps the `Action` in a `SecureAction`, which uses the JAAS `AccessController` to check the access for invoking the requested `Action`.
9. Since the actual `Action` is invoked by the `execute` method, the `AccessController` uses a `MethodInvokePermission` in order to check if the call to the `execute` method is allowed for the currently logged on user.
10. In order to determine if the currently logged user has the permission to invoke the `execute` method of the actual `Action`, the JAAS system checks the current policy, if the `MethodInvocationPermission` was granted to the currently logged-in user.
11. If the permission check was successful, finally, the `execute` method of the actual `Action` is invoked, and the results are returned to the user.

Although the procedure for authenticating and authorizing an command is rather complex due to the integration of JAAS, it provides great flexibility and interoperability. In fact, adapting the security configuration of the XMPP Log Management Framework to arbitrary JAAS authentication and authorization methods is simply achieved by configuring the appropriate JVM security properties.

JMX Plug-in

With the inclusion of the Java Management Extensions into the Java Standard Edition Version 5.0, the importance of providing a consistent and standardized management access to an application is increased. In order to integrate the proposed framework into JMX enabled systems, the XMPP Log Management Framework provides a basic JMX plug-in, the `JMX ExporterPlugin`. This plug-in, once enabled

in the log4j configuration, exposes all available log4j JMX components to a given MBeanServer. Using appropriate clients it is therefore possible to monitor and alter to a certain extent the log4j configuration for a given application. This way, the administrator has the choice whether to use JMX for log4j configuration or the XMPP CommandPlugin described in section 4.2.1.

4.2.2 Chainsaw Integration

In the previous chapters the design of the processing core of the XMPP Log Management Framework was discussed. In this chapter, the integration into Chainsaw, the graphical log viewer, will be outlined.

Chainsaw was originally designed as GUI front-end for log4j in order to display logged messages using a graphical user interface. A screenshot of the user interface is shown in figure 4.3.

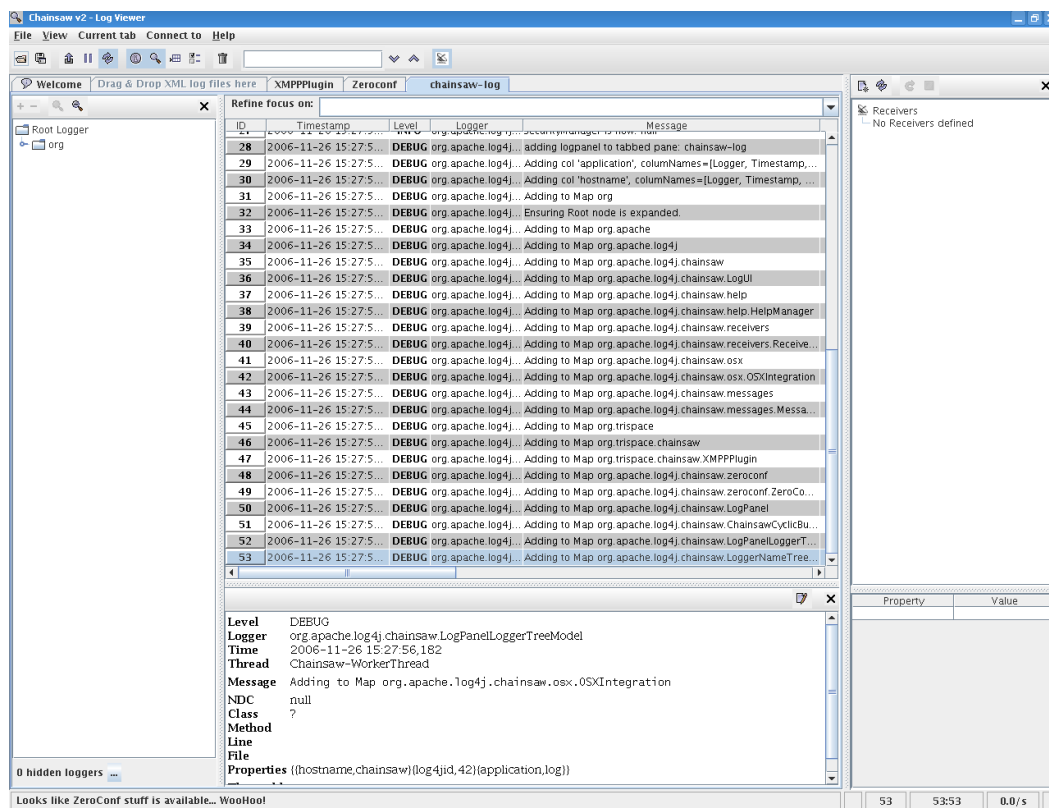


Figure 4.3: The Chainsaw log-viewer

As apparent from the screenshot, the Chainsaw main screen is built using 3 main components:

Message Panel The message panel forms the main component of the Chainsaw GUI where all received messages are displayed using in a list widget. According to the information contained in the actual received log messages, the

listview adapts its columns. By default, a serial number identifying the displayed messages, a timestamp, the level and originating logger, the message itself and an optional Java `Throwable` object are displayed.

In order to limit the amount of displayed messages, Chainsaw allows creating filters using comparison and boolean operators, like for example:

```
LEVEL > INFO && LOGGER == "org.trispace.MainLogger"
```

to filter out important messages from the `MainLogger` logging instance. To construct this filter expressions in a easy and intuitive way, Chainsaw allows generating them by clicking on the desired property and selecting “add to refine focus field”. In this way, a complex expression can be built step-by-step without the need to type the filter rules manually.

Logger Panel On the left a tree-like widget displays the runtime logger hierarchy. As discussed in chapter 4.2 the loggers form a hierarchy, with the root logger at the apex. Since the loggers can be dynamically requested by a running application, the visualization of the logger hierarchy is dynamically adapted to the running system.

By clicking at a leaf node of the hierarchy tree, Chainsaw highlights messages matching the selected logger in the main message panel.

Receiver Panel On the right side of the main screen, the receiver panel allows managing log4j receivers. Chainsaw uses log4j receivers in order to import log messages from systems or log4j instances outside the current Java virtual machine. In order to startup Chainsaw with a predefined set of receivers, it is possible to save and load the receiver configuration as log4j Joran-style configuration file.

Besides this three main panels, Chainsaw allocates tabbed panes for the integrated help system and additional plug-ins. This approach allows seamless integration of plug-ins which may require GUI components for configuration or for interacting with the user without overloading the user interface. Furthermore, Chainsaw allocates a new panel, when a initial message from a receiver plug-in is received, in order to concentrate all messages arriving from a specific receiver to a single panel.

The `XMPPPlugin`, for example uses a dedicated Chainsaw panel in order to provide a GUI for interacting with XMPP enabled log4j installations. The main goal of the `XMPPPlugin` is to integrate a component for controlling remote log servers into the Chainsaw application. The current version of the `XMPPPlugin` represents a prototype implementation of this goal. Currently it is possible to configure a list of servers including the needed user credentials and connect to them in order to execute commands. For each server the `XMPPPlugin` allocates a new panel, where all communication within the specific connection is handled.

4.3 A Real World Example

After having discussed the technical architecture of the XMPP Log Management Framework, I would like to demonstrate its flexibility and easy configuration by outlining a possible real world scenario:

A company would like to integrate data from an application running in a remote branch office and a locally installed legacy application into a newly deployed ERP system. Since the integration of the data was relatively easy to achieve, the logging of the concerned systems into a centralized log database is planned.

Since every component of the system uses a different logging implementation, the integration of their log data into a single repository requires a flexible logging framework, capable of integrating the following resources:

Log File Since the legacy system at the remote branch office logs into a simple text file, the content of this file needs to be integrated into the planned logging solution. In order to make this file available to the main site, it was placed on a network share from which the log management system accesses it.

Legacy Database Instead of logging to a file or using syslog, the locally installed legacy application appends its log messages to a MySQL database. The logging framework therefore needs to be capable of extracting those log messages from a database table.

Application Server The new ERP system at the main site is based on a J2EE application server, and outputs its log statements using the logging facilities provided by the application server, which in this case are based on log4j.

By analyzing the requirements it is evident that the classic architecture of a log management system, namely consisting of input, processing and output modules is highly applicable to this example. In fact in terms of the XMPP Log Management Framework, the task of receiving log messages from the concerned systems is typically implemented by log4j receiver modules. The processing and visualization of the received messages is handled by Chainsaw and the storage back-end is delegated to log4j appender modules.

Thanks to the flexible configuration of the XMPP Log Management Framework, the realization of the given example is mainly a matter of configuring the appropriate log4j receivers, respectively appenders. In the XMPP Log Management Framework this configuration is carried out by editing the main Chainsaw configuration file, since Chainsaw serves as main module of the framework, implementing the message processing and configuration logic.

The Chainsaw configuration file is fully compatible with the log4j XML “Joran” configuration format and has typically the following structure:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE log4j:configuration >
<log4j:configuration xmlns:log4j="http://jakarta.apache.org/log4j/">

<plugin name="pluginname" class="pluginclass">
```

```

    <!-- plugin specific configuration -->
</plugin>

<appender name="appendername" class="appenderclass">
    <!-- appender specific configuration -->
</appender>

<!-- definition of the root logger -->
<root>
    <level value="debug"/>
</root>

</log4j:configuration>

```

First we define a so called “LogFilePatternReceiver” receiver in order to extract messages from the legacy system out of its log file. This receiver monitors a given file and converts the appended log statements to log4j log objects, which are then handled as if directly produced by the log4j logging API.

In order to specify the details of this conversion step it is necessary to provide the pattern the legacy application uses for its log statements. Since receivers are typically implemented as Chainsaw plug-ins, we define them using a plug-in section:

```

<plugin name="fileReceiver"
class="org.apache.log4j.varia.LogFilePatternReceiver">
    <param name="fileURL" value="file:///share/branchoffice.log"/>
    <param name="timestampFormat" value="yyyy-MM-dd HH:mm:ss,SSS"/>
    <param name="logFormat" value="TIMESTAMP LEVEL CLASS - MESSAGE"/>
    <param name="name" value="branchOffice"/>
    <param name="tailing" value="true"/>
</plugin>

```

This configuration fragment defines a new LogFilePatternReceiver which monitors the file “branchoffice.log” and interprets appended lines with the pattern specified by the “logFormat” parameter.

The next step is to integrate the log data contained in the database used by the legacy application installed in the head office. By using a “CustomSQLDBReceiver” it is possible to integrate all databases supported by the Java JDBC database interface into log4j. Just like the receiver used to import log messages from a text file, also the database based receiver is implemented as plug-in and configured by the following configuration:

```

<plugin name="dbreceiver"
class="org.apache.log4j.db.CustomSQLDBReceiver">
<connectionSource
class="org.apache.log4j.db.DriverManagerConnection Source">
<param name="driverClass"
value="com.mysql.jdbc.Driver"/>

```

```
<param name="password" value="pass"/>
<param name="user" value="user"/>
<param name="url"
value="jdbc:mysql://legacyserver/sap"/>
</connectionSource>
<param name="IDField" value="logid"/>
<param name="refreshMillis" value="3000"/>
<param name="sql" value="SELECT log_id,
application as APPLICATION,
log_level as LEVEL, message as MESSAGE,
hostname as HOSTNAME, log_time as TIMESTAMP,
'' AS EXCEPTION, logger AS LOGGER
from logs"/>
</plugin>
```

After defining the required class name and connection URL of the MySQL database driver and providing user credentials in order to access the database, the plug-in is configured with the name of the column containing a unique identifier. In our case the database field “logid” is used to detect already extracted log messages between two database queries (which in our configuration are carried out every 3 seconds). Finally, the parameter “sql” defines the SQL command used to extract the needed information as well as the mapping of the database fields to log4j log object properties.

In order to further include the native log4j log messages generated by the ERP application, a simple logger is defined in the Chainsaw configuration:

```
<logger name="com.company.erp.main">
  <level value="info"/>
</logger>

<root>
  <level value="debug"/>
</root>
```

By defining the root logger (in our case with log level “debug”), the configuration is complete and Chainsaw is able to receive messages from the defined receivers. By configuring a database appender as follows:

```
<appender name="logstore"
class="org.apache.log4j.jdbcplus.JDBCAppender">
  <param name="url" value="jdbc:mysql://localhost/logs" />
  <param name="username" value="username" />
  <param name="password" value="username" />
  <param name="sql" value="INSERT INTO LOGTEST
(id, prio, cat, thread, msg)
VALUES (@INC@, '@PRIO@', '@CAT@', '@THREAD@', '@MSG@')" />
  <param name="dbclass" value="com.mysql.jdbc.Driver" />
</appender>
```

and changing the root logger statement to

```
<root>
  <level value="debug"/>
  <appender-ref ref="logstore" />
</root>
```

the log data from all appenders gets stored into the specified database for reliable long term storage.

This example uses only the basic capabilities of the XMPP Log Management Framework. However, for using XMPP as transport mechanism for log messages sent from remote locations, the configuration can be simply enhanced by placing a XMPPAppender at the log source and a XMPPReceiver at the central log repository. By using these XMPP based components, it is possible to encrypt the transmission of log messages and by enabling a XMPPCommandPlugin in the configuration, the administration of the log management system is possible through a XMPP based admin interface as outlined in chapter 4.2.1.

4.4 Summary

In this chapter, the novel approach of extending log4j in order to create a flexible and distributed log management framework was presented. The idea, to use XMPP as transport protocol for log data and administrative commands lead to the prototypical design of the XMPP Log Management Framework. After introducing its foundation, the Java based log framework log4j, the system design of the proposed framework was outlined and improvements over existing systems were illustrated. In order to demonstrate its flexible configuration scheme, the possibility to integrate legacy systems, as well as the extensible plug-in system, a real world example was introduced. In this example, the XMPP Log Management Framework was employed to integrate log messages from two legacy applications and a recent ERP application into a centralized, database backed log repository.

5 Logging in Event-Driven Environments

5.1 Introduction

After discussing the basic architecture of current log management systems and presenting the XMPP Log Management Framework, in this chapter a possible next step in the evolution of log management will be introduced: the integration of logging in event driven environments. After introducing the principles of event-driven systems, the advantages arising from the usage of such systems in order to process log messages will be outlined. Furthermore, advanced message processing algorithms specifically used in event-driven environments are presented.

A event-driven architecture denotes a system architecture which is based on the reception, processing and sending of events, where an event is typically defined as:

“A notable activity that happens.” [Luc06]

In a event-driven system, an event can be for example, a change of state in a state machine, an occurrence of an error, or the raise of temperature on an external sensor.

In event-triggered systems, a systems action is *triggered* by the reception of an event, which was generated at a *source* and is transmitted to a *target*. This process of receiving, evaluating and acting upon an event is typically denoted as *event processing*.

Generally, event processing is categorized into the following three styles:

Simple Event Processing: Simple event processing forms the most basic type of event processing, where the occurrence of an event triggers one or more downstream actions [Mic06].

Stream Event Processing: Stream event processing is typically deployed in order to process real time event flows. According to Brenda Michelson [Mic06] “In stream event processing, both ordinary and notable events happen. Ordinary events (orders, RFID transmissions) are both screened for notability and streamed to information subscribers”.

complex event processing: Events that are an abstraction of two or more events are denoted as *complex events*¹ [Luc06]. Such abstraction is typically of spatial, temporal or causal nature. Complex event processing deals with the process of evaluating such events and derive actions based on the evaluation results.

¹often also denoted as *composite events*

In the area of realtime analysis of business processes especially the stream event processing and particularly the complex event processing have gained lot of attention in the last years. With the introduction of loosely coupled system architectures, such as the service oriented architecture (SOA), event processing was an ideal accompanying technology. Since event-based systems are typically loosely coupled by its nature they can be easily integrated in service oriented architectures. In such environments, the occurrence of an event is handled by an event processing system which in turn triggers the invocation of one or many services. [Mic06].

5.2 Log Message Processing in Event-Triggered Environments

Since log message processing can be seen as use case of event processing, the integration of log message processing into an event-triggered system is promising. In fact, the point in time, when an application or operating system emits a log message could be seen as event in an event-triggered architecture, which may be further processed by various event processing algorithms.

As outlined by David Lukam in [Luc06], an event, or more specifically an *event object*, which represents an occurred event in a computing system is typically decorated with *event attributes*. This event attributes enrich the event with extended properties, like for example an unique identifier or a timestamp.

Speaking of the integration of logging in event-triggered architectures, a log event object could be, for example, decorated with event attributes containing the detailed log message, a field containing the severity or any other records as discussed in chapter 1.2.1, describing the basic structure of a log message. This way, an originally decoupled logging subsystem can be easily integrated in an existing event-triggered architecture, possibly so far only utilized for processing business events like order placements or machine state changes.

It is important to note that logging is typically considered as tool to ease debugging and optimizing a computing system only. This assumption is fortified by the matter of fact, that current systems rarely integrate the information contained in log files into business processes. However, in many cases the information extracted from the log data analysis could provide valuable auxiliary information for a specific business case.

For example, in a automated production facility a command for building a specific good is typically sent to a chain of machines and results into a “production complete event” after successful production. Typically only the initial start command and the this final completion event is incorporated into a business management system. However, as the product is manufactured step-by-step by a number of subsystems and machines, logging on each machine could provide valuable information about the flow of the product throughout the production facility. By analyzing this data it is for example possible to optimize the path or execution order of the different manufacturing steps in order to increase the throughput of the whole system and avoid delays caused by uncoordinated production work flows. While this approach uses long-term recorded data in order optimize a work flow, when combining logging with event-triggered architectures this optimization and adaption to changed

requirements could be carried out continuously and on the running system.

By analyzing the logged data using an event-triggered system, valuable information contained in the log message, which typically includes detailed information about each process step could be taken into account for further business decisions. Although this approach represents a rather new use case for log data extraction, this example shows the power behind the combination of both technologies.

The integration of logging in event-driven architectures can therefore be regarded as next step in the evolution of log management systems:

The Log Message as Operational Value In conventional logging systems, a single log message has typically no or little *operational value*: As said, in such systems, a log message is generally used only for statistical, debugging or archiving purposes. Although the recorded messages might be interpreted at some later point in time, at the actual time of reception, the message is typically not of great importance. In fact, a possible operational value is typically gained only at that later point in time, when the collected log messages are interpreted by a log analyzing tool or by the administrator itself.

The combination of logging with event-triggered architectures however changes this behavior: The reception of a log message is regarded as event, which, together with the information contained in the received message is typically *immediately* handled by the event processing system. Therefore, by including the event of log message reception into the event processing system, the log message gains an immediate operational value. This operational value is therefore available nearly immediately after reception of the message rather than after a separately conducted log data analysis.

The interpretation of the log message reception as event therefore facilitates immediate reaction and integrates the logging system seamlessly into a event driven architecture.

Typically, such event processing systems are implemented using stream event processing or complex event processing architectures, where state changes reported by the logging system need to be evaluated in realtime and notable events should trigger certain actions.

Logging as Integration Technology The proposal of including logging into event-triggered systems leads to another possible use case: the integration or migration of legacy systems into event-driven environments. By adapting the logging subsystem of the legacy system to emit log messages on defined events, these log messages could be easily integrated into a event-driven environment, like outlined in the previous paragraphs. As most legacy systems use dedicated logging libraries, such as log4j, the logging subsystem is decoupled from the application itself. By adding modules to the logging library, in order to emit an event upon a log request from the application, the legacy system typically remains unchanged, which is often the main requirement in migration projects. But still, the legacy application is now able to send events to the event-driven system.

Although other, more efficient methods for connecting legacy systems with new architectures, such as the mentioned event-driven architectures may exist, this ex-

ample still shows the broad range of use cases a modern logging system has to offer. Furthermore it encourages a new view at the logging process and log data itself, by illustrating new ways of log data semantics and log data processing.

By looking at the presented examples and advantages, integrating logging sub-systems into complex event processing (CEP) architectures is a promising and important goal in the design of future log management systems. One of the main features of CEP, the possibility to infer a complex event out of a number of simple events is indeed very useful for log data processing. As mentioned in chapter 2.3.3, event correlation is a key requirement for recent log management systems in order to extract important information out of ordinary data.

Utilizing complex event processing systems in order to identify and correlate notable events out of log data streams and generate a representing complex event for further processing is probably the key feature for a new type of log management systems, which are fully integrated in loosely coupled architectures like the mentioned event-triggered systems.

This integration has also the advantage that with the improvement of current event-driven systems and their event processing algorithms also the processing of log data enhances. Furthermore, the clear separation of concerns between the log storage and log management by using traditional log management infrastructures for managing and storing log data and event-driven architectures for analyzing and processing the logged data results typically in a cleaner system design.

5.3 Advanced Message Processing and Root Cause Analysis

After the logged data has been saved to a persistent storage, a main requirement for a log management system is to analyze the logged data, in order to provide input for optimization, debugging or statistical purposes. Especially debugging activities require the ability to extract specific events out of possibly multiple log files from various systems and merge them into a sorted sequence of consecutive events. Then, in this so obtained streams, correlated events need to be identified. This task is particularly complex if the logged dataset contains events generated from multiple sources, possibly linked to each other. Typically in such environments the occurrence of an event in a single system entails events on other systems as well. While such behavior is commonly accepted and more often also wished, it can introduce however unexpected complexity and error behaviors.

In order to illustrate the problem, I would like to present the following scenario:

A web-based ordering system for a company selling goods over the Internet, where the technical infrastructure consists of the following components:

Web Server: The Web-server, running a servlet container, implements the business logic for the order system. The servlet queries the database and renders the results as HTML page. Once the user has chosen the article of his interest, the user interface allows to place an order, where the shipping address and an email address for validating the customer needs to be provided.

Database Server: The database server stores the prices, description and availability of the goods as well as the current orders.

Logging System: A central log server is used in order to collect all log data from the concerned systems in a central repository.

Lets now consider the situation where the database server is down for a few minutes while a customer is about to enter his shipping information and email address.

The Web application returns an error, indicating that the verification of the email address failed (without giving any further details) and logs this error. After trying to refresh the page, the Web application makes a database query in order to reread the database contents which fails, since the database server is down. This failure gets logged too. The customer calls the companies support center, complaining about the problem its email address cannot be verified. As consequence the IT-staff tries to find the problem by inspecting the log files. Since the Web application was the first to log about a failed email verification, the support staff suspects the fault in the Web application, rather identifying that the *root cause* of the problem was not the Web application but the non functioning database server, which lead to the misleading error message of email address verification problems.

Although this example might be a little bit oversimplified, the important aspect here is that the *root cause*, thus the actual event responsible for the system failure was not identified. Instead, another system component being affected by the side-effects of the root cause was wrongly suspected, leading to time consuming debugging activities. This shows the need for the integration of advanced log data analysis into current log management systems.

In event-driven architectures, the process of detecting the root cause, denoted as *root cause detection* is typically addressed by complex event processing algorithms. In this systems, the stream of events is constantly scanned in order to correlate notable events and generate a complex event if the correlation succeeds. Instead of signaling every notable event, only complex events, representing a certain sequence of notable events are forwarded to the other nodes in the event-driven system.

This approach, of merging a sequence of notable events into a complex event offers further the advantage that the amount of events transferred throughout the system is greatly reduced. Also the problem of analyzing log data from interconnected systems like clustered systems may benefit from complex event processing. Consecutive events or failures, as outlined in the example, can be identified as related - thus as single complex event. This approach exposes an important fact: the generated complex events² can now be seen as “events from the system”, rather than as events from the individual sub components. Such high-level view on a networked system is for example helpful in order to quickly identify the consequences caused by a system fault, or for gaining an impression about the current system activities grouped by functional units rather than individual machines.

The integration of logging and complex event processing further ensures, that *all* events are available in the log store, not only notable ones. Therefore, when detailed debugging information is needed, or when the results of the event-processing need to be verified, the log database can still be consulted. This fact, that all events served as basis for the deduction of complex events are available also after the processing step is an important requirement for dependable event-processing systems.

²in a networked environment

6 Evaluation and Future Work

In this chapter, possible future work in the field of log management and log data processing will be outlined, and possible solutions for current problems will be presented.

Log management frameworks and applications are nowadays well established components of nearly every computer system. While a few decades ago, storing data volumes in the region of several gigabytes was nearly impossible or at least very expensive, nowadays, with the availability of cheap mass-storage, enterprise storage systems are able to handle data volumes consisting of multiple terabytes. However, while storing such amounts of data is technically possible and well established, extracting relevant information out of the stored data is still problematic.

Algorithms like the event correlation algorithm discussed in chapter 2.3.3 are indeed useful for detecting logically coherent messages and thus useful for minimizing superfluous entries from a log data store. However, a main drawback of this algorithm is that the rules which define *how* messages correlate to each other need to be defined manually. Therefore, an a-priori knowledge of the to be analyzed data is needed in order to work with this algorithm. Although this knowledge is available in a large percentage of use cases, it is desirable to extract those rules from a running system without former definition. Therefore, in the future more advanced algorithms might be used in order to avoid the tedious and error-prone rule definition.

6.1 Advanced Message Correlation

As outlined in chapter 2.3.3, various algorithms for correlation of similar events are available. Typically algorithms as implemented for example in SEC [Vaa06b] use relatively simple ideas like a time window in order to correlate events. However, when dealing with large amounts of messages or when correlation over multiple data sets is desired, those simple correlation algorithms might not be appropriate. In this case more advanced algorithms like self learning systems or algorithms used in data mining applications might be a solution. Especially, the tedious and often error-prone rule definition is one of the main drawbacks of simpler algorithms, which can be avoided using self-learning systems. Rules could therefore be inferred automatically by analyzing the occurrence and type of events from a typical controlled operation and using irregular or sporadic events in order to dynamically create new logging rules or changing the current loglevel. However, while the utilization of such automatisms provides benefits in terms of lower maintenance costs, the sensitivity to denial of service attacks is increased considerably. It is therefore advisable to regard such algorithms as useful additions to traditional log processing algorithms.

6.2 Integration in Systems Management

In nearly all recent operating systems, the logging subsystem is more or less decoupled from other system components, such as the security infrastructure or other low-level system components. In this case, the framework's sole function is to reliably save received log messages to the systems storage. By looking at the above mentioned possibilities of extracting valuable information out of the logged data, integrating the log management system into, for example, a systems security infrastructure makes sense. An example of such system design is a intrusion detection system, which takes advantage of the data provided by the logging framework: By linking and integrating status or error information acquired from the logging framework into the decision logic, anomalies of the system behavior or unexpected status changes can be used as additional input in order to alert system administrators or trigger evasive actions as stated in [ATS⁺03]. Furthermore, by providing auxiliary information gained from the logging subsystem in addition to conventional sources like network traffic or agent data, the IDS core logic might be able to minimize the occurrence of false positives [ATS⁺03]. Also the overall reliability and robustness of an IDS can be increased using this approach, since a logging framework typically receives data from multiple sources, therefore not being affected by the loss of a single log source. Furthermore, by integrating the data gained from a distributed logging framework, the IDS could detect and react upon distributed attacks or denial of service attacks rather than relying on local network traffic analysis only. In this context, advanced processing algorithms like complex event processing are especially helpful for separating typical, less important events from unexpected ones.

For example, a rollout of a cluster system, consisting of various components, like database servers, application servers and other system components could be monitored by a complex event processing system. Log messages received by the mentioned system components could be processed in order to ensure that each system component works as expected. The inclusion of expected events into the event processing logic can be used to detect failed systems or unplanned startup sequences.

It is therefore clear, that integrating the data obtained from logging systems into systems management provides many advantages. Therefore, the evolution of log management systems might lead towards fully integrated logging solutions in order to provide valuable data for various use cases including security, accounting and statistical purposes.

7 Summary and Conclusion

Algorithms and system architectures for storing, managing and analyzing log data were for a long time not amongst the most discussed topics in the world of information technology. This circumstance however changed rapidly with the establishment of the various network technologies and particularly with the introduction of the Internet and large scale data warehouses in various businesses and organizations. Suddenly, the information contained in application and system logs was not only useful for debugging and optimization purposes, but gained the attention of the financial and controlling staff. Questions like “which sites are the top visited ones?” or “where and how do users find our site?” can be easily answered by analyzing the various log files of web and application servers.

While nowadays companies like Google and Amazon use logfile-based data mining extensively, all these use cases are only possible with an effective and reliable log management. As mentioned in the introduction chapter, the basic logging facilities provided by the operation systems are more and more replaced or extended by powerful log management systems.

Considering that these types of applications are generally not long-established in current system environments it is advantageous to gain an overview of possible implementations and features. This thesis has provided fundamental descriptions of the various components and provided an overview of the requirements a log management system must fulfill.

By introducing a framework for a log management system based on a widely adopted logging framework and providing an example how such system could be integrated in a instant messaging network, a novel concept of designing a system for collecting and managing logged data was presented. The key feature of the presented log management framework, its extensive modularity, is an example how rather inhomogeneous system components can be integrated into a complete and adaptable framework.

Furthermore, techniques and algorithms for analyzing the logged data and the various possibilities of integrating their results in event-based business decision systems were outlined and examples of their usage were discussed. With an outlook on how event based log processing could possibly lead to a new generation of log management systems, this thesis further shows how the role of logging may change from a tool for debugging and reporting purposes only towards a source of valuable information for a variety of use cases.

A log management system can therefore be seen as an important system component, providing usage and system status details over far-reaching time spans needed for planning, extending and managing effective and reliable computer systems.

7 Summary and Conclusion

Appendix

A Syslog

syslog is the name for a network protocol, the *syslog protocol*, and the library implementing the syslog protocol. It was developed by Eric Allman as part of the sendmail project and is nowadays the de facto standard for handling log messages on UNIX and UNIX-like systems. The protocol itself is not standardized by the IETF but documented in RFC 3164 [Lon01].

Besides the ability to record log messages from the machine itself, syslog allows to receive and forward messages from, respectively to other syslog daemons over the network.

A syslog log message contains the following records:

NAME	DESCRIPTION
PRI	Numerically encoded facility and severity information. Facilities represent the source application or system process of the log message. The RFC defines 24 facilities such as kernel, mail subsystem, security subsystem etc. The severity is used to indicate the level of the message. 8 severities ranging from "emergency" to "debug" are defined in the RFC.
HEADER	The header contains a timestamp and a field including the hostname or IP address of the sending host. The sending host may be the local host, or a host on the network
MSG	This record contains additional textual information which is provided by the logging application itself

Table A.1: The format of a syslog log message

Generating such syslog messages is typically accomplished by using the syslog library which passes them to the syslog daemon (typically called syslogd).

The syslog daemon is a UNIX daemon process listening on a domain or UDP socket for incoming messages and processes them according to instructions given in the syslogd configuration file. These instructions typically configure the routing of messages based on the priority field to different files or remote loggers.

The ability to receive messages via UDP allows creating centralized logging servers. This is mainly useful if the local system lacks storage capacity for collecting its own log messages or if a centralized repository for log messages is desired.

Although its vast dissemination the syslog protocol has some major drawbacks:

Limitation of message length The RFC defines a maximum message length of 1024 bytes for a syslog message. This often leads to problems when logging

extended information such as large stack traces or other extensive diagnostic messages.

Use of UDP as transfer protocol Although using UDP as transfer protocol for syslog messages imposes little overhead to the actual transferred messages, the stateless nature of the UDP protocol does not allow to have the reception of the message to be acknowledged at protocol level. Thus, messages sent through syslogd are typically transferred with “best-effort” without any guarantee or confirmation of a successful delivery.

B NT Event Log

Like UNIX systems, also the Windows operating system contains a logging framework. It is called “event log” [win06] and built upon three main components:

- Event Logging Service
- Event Logging API
- Event Viewer

The “Event Logging Service” serves as foundation for the Windows event logging framework. It is implemented as a service (contained in the executable file “eventlog.exe”) and gets started by the service control manager (SCM) on system startup. To store received logging events the event logging services maintains (in its default configuration) 3 log files:

Application Event Log File This log file (typically called “apptevent.evt”), records logging events received from user applications.

Security Event Log File This log file (typically called “secevent.evt”) is used to record security related events like user authentications or other events received from the security subsystem.

System Event Log File This, typically “sysevent.evt” called log file is used to keep track of log messages generated from the Windows core itself, like messages from the service control manager, the network subsystem etc.

These logging files are binary encoded and contain support for internationalization. The internationalized messages and captions are not stored in the log file itself, but references to files outside the event log files.

To access the eventlog itself, an application uses the eventlog API. This application programming interface allows creating and reading log events and manage the logging system. The access is not restricted to the local system only, but also through the network using RPC, where further details dealing with the network connection are encapsulated by the API, so that accessing a remote eventlog instead of the local one differs only by the usage of UNC-style paths for specifying the remote machine.

Like almost every system component in Windows, also the eventlog API makes use of the Windows security subsystem, allowing a administrator to set fine granular permissions to the eventlog, the API calls and the log files itself, thus restricting the access effectively and in an integrated manner for the local system as well as remote ones.

Although it is possible to define the maximum size of the generated log files, the Windows eventlog performs no further log file management like rotation or backup, but can be configured to overwrite older entries or deny further reception of log events if the file reaches a specified limit.

An example of an application using the managing facilities provided by the eventlog API is the Windows built-in Event Viewer. This application may be used to browse the above mentioned log files or display auxiliary evt-files. Furthermore it is possible to limit the displayed data with filters or access the eventlog from remote computers.

The format of a Windows eventlog entry is comparable to an Unix syslog entry, but provides some extensions as stated in table B.1.

NAME	DESCRIPTION
Type	Used to categorize the event. Common types include “information”, “error”, “warning”, thus making this element comparable to the syslog “PRI” element
Timestamp	The timestamp of the event entry
Source	Identifies the source of the event like “service control manager” or “browser service”
Category	This element may be used to categorize an event within an application.
Event ID	Used to identify the type of an event. Each application may define an uniquely event ID for each possible event.
User	The owner name of the generating application
Computer	The name of the computer who generated the event
Description	Descriptive text of the event
Data	Additional data - usually used for debugging purposes - formatted as plain-text or binary

Table B.1: The format of a Windows eventlog entry

In order to group event with correlated type together, the Windows Eventlog defines the following base categories, as defined in [win06].

Error: “An event that indicates a significant problem such as loss of data or loss of functionality”

Warning: “An event that is not necessarily significant, but may indicate a possible future problem”

Information: “An event that describes the successful operation of an application, driver, or service”

Appendix

Success Audit: “An event that records an audited security access attempt that is successful”

Failure Audit: “An event that records an audited security access attempt that fails”

Bibliography

- [AA01] ALGIRDAS AVIZIENIS, Brian R. Jean-Claude Laprie L. Jean-Claude Laprie: Fundamental Concepts of Dependability / Laboratoire d'Analyse et d'Architecture des Systemes. Version: 2001. <http://www.cert.org/research/isw/isw2000/papers/56.pdf>. 2001. – Techreport
- [Ada02] ADAM, Sah: A New Architecture for Managing Enterprise Log Data. In: *Proceedings of the 16th Systems Administration Conference (LISA-02)*. Philadelphia, Pennsylvania : USENIX Association, 2002, 121–132
- [apa06] The Apache Software Foundation: *Apache Logging Services*. <http://logging.apache.org/>. Version: 2006
- [APS99] ALLMAN, Mark ; PAXSON, Vern ; STEVENS, W. R.: *TCP Congestion Control*. <http://www.ietf.org/rfc/rfc2581.txt>. Version: 1999 (Request For Comments)
- [ATS⁺03] ABAD, Cristina ; TAYLOR, Jed ; SENGUL, Cigdem ; YURCIK, William ; ZHOU, Yuanyuan ; ROWE, Ken: Log Correlation for Intrusion Detection: A Proof of Concept? In: *19th Annual Computer Security Applications Conference (ACSAC)*. Las Vegas, Nevada, 2003, 255-264
- [Bon00] BONDI, André B.: Characteristics of scalability and their impact on performance. In: *Proceedings of the 2nd international workshop on Software and performance*. Ottawa, Canada : ACM Press New York, NY, USA, 2000, 195 - 203
- [Bur02] BURN, Oliver: *Chainsaw Log Viewer*. <http://logui.sourceforge.net/>. Version: 2002
- [BWNH⁺06] BLAKE-WILSON, Simon ; NYSTROM, Magnus ; HOPWOOD, David ; MIKKELSEN, Jan ; WRIGHT, Tim: *Transport Layer Security (TLS) Extensions*. <ftp://ftp.rfc-editor.org/in-notes/rfc4366.txt>. Version: 2006 (Request For Comments)
- [CKC05] CALLAS, Jon ; KELSEY, John ; CLEMM, Alexander: *Signed syslog Messages*. <http://tools.ietf.org/html/draft-ietf-syslog-sign>. Version: May 2005 (Internet-Draft)
- [com06] The Apache Software Foundation: *Common Log Format*. <http://httpd.apache.org/docs/2.2/logs.html#accesslog>. Version: 2006

Bibliography

- [dep06] International Federation for Information Processing: *Dependable Computing and Fault Tolerance*. <http://www.dependability.org/wg10.4/>. Version: 2006
- [Des07] DESTAILLEUR, Laurent: *Free Real-Time Logfile Analyzer To Get Advanced Statistics*. <http://www.awstats.net/>. Version: 2007
- [Dij03] DIJKSTRA, Edsger W.: *On The Role of Scientific Thought*. <http://www.cs.utexas.edu/users/EWD/transcriptions/EWD04xx/EWD447.html>. Version: 2003
- [DR06] DIERKS, Tim ; RESCORLA, Eric: *The TLS Protocol Version 1.2*. <http://tools.ietf.org/html/draft-ietf-tls-rfc4346-bis>. Version: 2006 (Internet-Draft)
- [Eat03] EATON, Ian: *The Ins and Outs of System Logging Using Syslog / The SANS Institute*. Version: 2003. <http://www.sans.org/rr/whitepapers/logging/1168.php>. SANS Institute, 2003. – Techreport
- [eu-05] The European Parliament: *Directive Of The European Parliament And Of The Council on the retention of data generated or processed in connection with the provision of publicly available electronic communications services or of public communications networks and amending Directive 2002/58/EC*. <http://register.consilium.eu.int/pdf/en/05/st03/st03677.en05.pdf>. Version: 2005
- [Ger05] GERHARDS, Rainer: *The syslog Protocol*. <http://tools.ietf.org/html/draft-ietf-syslog-protocol>. Version: October 2005 (Internet-Draft)
- [GLS⁺02] GONÇALVES, Marcos A. ; LUO, Ming ; SHEN, Rao ; FAROOQ, Mir ; FOX, Edward. A.: *An XML Log Standard and Tool for Digital Library Logging Analysis*. In: *Proceedings of the Sixth European Conference on Research and Advanced Technology for Digital Libraries*. Rome, Italy : Springer-Verlag, 2002, 129–143
- [HBB96] HALLAM-BAKER, Phillip M. ; BEHLENDORF, Brian: *Extended Log File Format*. <http://www.w3.org/TR/WD-logfile.html>. Version: 1996 (W3C Working Drafts)
- [iee96] Institute of Electrical and Electronics Engineers: *The IEEE Standard Dictionary of Electrical and Electronics Terms*. 1996
- [ISO98] International Organization for Standardization: *Information Technology – Open Systems Interconnection – Distributed Transaction Processing, Part 1: OSI TP Model*. 1998
- [ISO05] International Organization for Standardization: *Information Technology. Code of Practice for Information Security Management*. 2005

- [jaa06] Sun Microsystems, Inc.: *Java Authentication and Authorization Service*. <http://java.sun.com/products/jaas/>. Version: 2006
- [jsf06] *Jabber Software Foundation*. <http://www.jabber.org/jsf/>. Version: 2006
- [JW95] JAKOBSON, Gabriel ; WEISSMAN, Mark: Real-Time Telecommunication Network Management: Extending Event Correlation With Temporal Constraints. In: *Proceedings of the fourth international symposium on Integrated network management IV*. London, UK : Chapman & Hall, Ltd., 1995, S. 290–301
- [KC81] KAHN, Robert E. ; CERF, Vinton G.: *Transmission Control Protocol*. <http://www.ietf.org/rfc/rfc793.txt>. Version: 1981 (Request For Comments)
- [KC93] KAHN, Robert E. ; CERF, Vinton G.: *Federal Information Security Management Act*. <http://csrc.nist.gov/sec-cert/>. Version: 1993
- [KM06] KARSH, Bruce ; MILLER, Bob: *fam - File Alternation Monitor*. <http://oss.sgi.com/projects/fam/>. Version: 2006
- [KS06] KENT, Karen ; SOUPPAYA, Murugiah: *Guide to Computer Security Log Management*. <http://csrc.nist.gov/publications/nistpubs/800-92/SP800-92.pdf>. Version: 2006 (NIST Special Publication)
- [Lau07] LAURENT, Domisse: *W3perl, a GPL Web Stats Analyser*. <http://www.w3perl.com/softs/>. Version: 2007
- [LE04] LEY, Wolfgang ; ELLERMAN, Uwe: *logsurfer*. <http://www.cert.dfn.de/eng/logsurf/>. Version: 2004
- [log06] The Apache Software Foundation: *Apache log4j Project*. <http://logging.apache.org/log4j/docs/index.html>. Version: 2006
- [Lon01] LONVICK, Chris: *The BSD Syslog Protocol*. <http://www.ietf.org/rfc/rfc3164.txt>. Version: August 2001 (Request for Comments)
- [Luc06] LUCKHAM, David: *Event Processing Glossary - Take 13*. <http://complexevents.com/?p=124>. Version: 2006
- [Luo95] LUOTONEN, Ari: *The Common Logfile Format*. <http://www.w3.org/pub/WWW/Daemon/User/Config/Logging.html>. Version: 1995 (W3C Working Drafts)
- [Mar02] MARLBOROUGH, Brad: *LogFactor5 Donation*. <http://blog.gmane.org/gmane.comp.jakarta.log4j.devel/day=20020411>. Version: 2002
- [Mic06] MICHELSON, Brenda: *Event-Driven Architecture Overview*. http://elementallinks.typepad.com/bmichelson/2006/02/eventdriven_arc.html. Version: 2006

Bibliography

- [Moo97] MOONEY, James D.: *Bringing Portability to the Software Process*. http://www.cs.wvu.edu/~jdm/research/portability/reports/TR_97-1.pdf. Version: 1997
- [Mye97] MYERS, John G.: *Simple Authentication and Security Layer (SASL)*. <http://www.ietf.org/rfc/rfc2222.txt>. Version: 1997 (Request for Comments)
- [NR01] NEW, Darren ; ROSE, Marshall T.: *Reliable Delivery for syslog*. <ftp://ftp.rfc-editor.org/in-notes/rfc3195.txt>. Version: November 2001 (Request for Comments)
- [Par01] PARK, Don: *XLF, Extensible Log Format*. <http://xml.coverpages.org/xlf.html>. Version: 2001
- [Pre03] PREWETT, James E.: Analyzing Cluster Logs Using Logsurfer. In: *Proceedings of the 4th Annual Conference on Linux Clusters*. San Jose, California, 2003
- [Pun01] PUNIN, John: *LOGML (Log Markup Language)*. <http://www.cs.rpi.edu/~puninj/LOGML/>. Version: 2001
- [Ram04] RAMSDELL, Blake: *Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 3.1 Message Specification*. <http://www.ietf.org/rfc/rfc3851.txt>. Version: 2004 (Request for Comments)
- [SA04a] SAINT-ANDRE, Peter: *End-to-End Signing and Object Encryption for the Extensible Messaging and Presence Protocol (XMPP)*. <http://www.ietf.org/rfc/rfc3923.txt>. Version: October 2004 (Request for Comments)
- [SA04b] SAINT-ANDRE, Peter: *Extensible Messaging and Presence Protocol (XMPP): Core*. <http://www.ietf.org/rfc/rfc3920.txt>. Version: October 2004 (Request for Comments)
- [SA04c] SAINT-ANDRE, Peter: *Extensible Messaging and Presence Protocol (XMPP): Instant Messaging and Presence*. <http://www.ietf.org/rfc/rfc3921.txt>. Version: October 2004 (Request for Comments)
- [sen06] SensAge: *SensAge Enterprise Security Analysis*. http://www.sensage.com/English/Products/SenSage_ESA.html. Version: 2006
- [Shi00] SHIREY, Robert W.: *Internet Security Glossary*. <http://tools.ietf.org/html/rfc2828>. Version: 2000 (Request For Comments)
- [Tho] THOMPSON, Kerry: *logsurfer+*. <http://www.crypt.gen.nz/logsurfer/>
- [TS02] TANENBAUM, Andrew S. ; STEEN, Maarten van: *Distributed Systems, Principles and Paradigms*. Prentice Hall, 2002

- [TTKH02] TAKADA ; TETSUJI ; KOIKE ; HIDEKI: MieLog: A Highly Interactive Visual Log Browser Using Information Visualization and Statistical Analysis. In: *Proceedings of the 16th Systems Administration Conference (LISA-02)*. Philadelphia, Pennsylvania : USENIX Association, 2002, 133–144
- [Vaa05] VAARANDI, Risto: *Tools and Techniques for Event Log Analysis*, Tallinn University Of Technology, Thesis, 2005. <http://kodu.neti.ee/~risto/publications/thesis.pdf>
- [Vaa06a] VAARANDI, Risto: *SEC - Simple Event Correlator*. <http://kodu.neti.ee/~risto/sec/>. Version: 2006
- [Vaa06b] VAARANDI, Risto: Simple Event Correlator for Real-Time Security Log Monitoring. In: *Hakin9 Magazine* 1/2006 (2006), 28–39. http://en.hakin9.org/attachments/pdf/hakin9_05_2006_10_EN_str28-39.pdf
- [vis06] Microsoft Corporation: *About Windows Event Log*. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wes/wes/about_the_windows_event_log.asp. Version: 2006
- [Wei01] WEICK, Karl E.: *Making Sense of the Organization*. Blackwell Publishing Ltd, 2001
- [win06] Microsoft Corporation: *About Event Logging*. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/eventlog/base/about_event_logging.asp. Version: 2006