Journal of Computer Science, 9 (1): 74-82, 2013
ISSN 1549-3636

doi:10.3844/jcssp.2013.74.82 Published Online 9 (1) 2013 (http://www.thescipub.com/jcs.toc)

Collecting Data from Running Systems

[1]Jess Nielsen and [2]Sufyan Almajali

[1]Department of Research and Development, Trapeze Group Europe A/S,
Søren Frichs Vej 38K, DK-8230 Åbyhøj, Denmark
[2]Department of Computer Science, The King Hussein School for Information Technology,
Princess Sumaya University, Amman, Jordan

Received 2012-11-12, Revised 2013-01-18; Accepted 2013-02-12

ABSTRACT

This article describes a way to build a consolidated log that can be used to facilitate the state of Windows services at runtime. It makes a side by side introduction of code instrumentation techniques and autonomic techniques that can be used to collected data from running systems in order to construct a consolidated log. The techniques try to deal with the non-standardized syntaxes and contents that come with heterogeneous logs. The collected data might be a foundation for a replacement to such logs hence the behavior can be determined by such collected data. These behaviors can be mapped to common, standardized and easy-understandable high level events that can be used as a log themselves.

Keywords: About Autonomic Computing, Code Instrumentation, Runtime Behaviors, Windows Services

1. INTRODUCTION

This Document outlines an approach that can be used to clarify the behavior of Windows services based on potential problems when localizing log information in an IT infrastructure.

Today's systems management is characterized by many systems that are monitored and managed individually, which is based on our own observations and experiences from previous jobs inside systems management, where surveillance and maintenance of mission-critical systems are the most recent tasks.

Each of the systems owns their own logs which typically have their own syntaxes and even semantics. This requires the IT staff to achieve special knowledge within each system and it makes it difficult to compare the correct logs and analyze them. The primary motivation is therefore to make it easier for the systems management staff through a standardized and easy-understandable consolidated log, where possible. An example of such log can be found in **Fig. 1**.

Consolidation (or data consolidation) is the procedure through which you install a central data storage unit to keep all databases easily accessible. Consolidation is not unlike data center migrations but the focus is on having the files or data in a specified location rather than actually moving them around.

The introduction of a consolidated log has the goal to save valuable time. The reason for this is that it prevents the search of logs in various locations and analysis of wrong logs by conceptually placing all log information at one place. The knowledge of different systems and particularly their proprietary error codes is no longer needed when all information is expressed in a standardized and easy-understandable way.

1.1. An Autonomic Approach

Consolidated logs address autonomic systems through the second level, which is also called the managed level, on the autonomic axis (IBM, 2005) with responsibility for data consolidation. This is an obvious reason to introduce autonomic systems. Along with consolidation, adding autonomic-like features becomes possible. Automatically triggering the action "restart system" on certain high level events is an example of this.

Corresponding Author: Jess Nielsen, Department of Research and Development, Trapeze Group Europe A/S,
Søren Frichs Vej 38K, DK-8230 Åbyhøj, Denmark

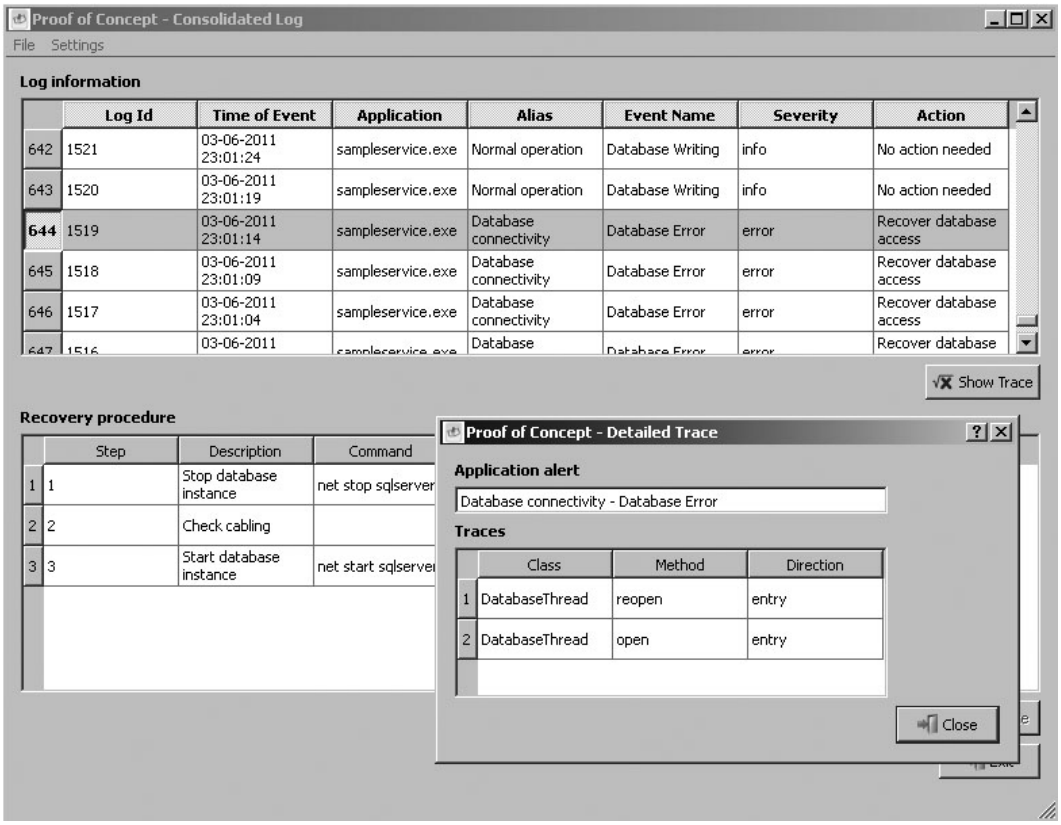

Fig. 1. A consolidated log

Table 1. The layers in the three-layered reference model

Sense	Act	Plan
Component control layer	Change management layer	Goal management layer
The bottom/operational layer	The sequencing layer	The uppermost/deliberation layer
Surveillance of the target systems	Execution of plans due to changes in the states	Planning in relation to high level goals and business policies
Reports states to the change management layer	Requests plans from the goal management layer	Provides plans to the change management layer

In general systems in the "autonomic world" are categorized into a five-level taxonomy that has been spearheaded by IBM. Each level represents a more advanced and refined system. The lowest level is a manual system with no explicit requirements for self-managed properties while the fourth and fifth level must have all of the self-* properties such as self-configuration, self-healing, self-optimization and self-protection (Kramer and Magee, 2007) to achieve their respective goals as being both decision making and business policy driven.

In general, software architecture defines the structure of the system as depicted in (Bass *et al*., 2003) and this is the basis for both manual and self-managed systems, but the self-managed systems state some explicit requirements to achieve the autonomic features.

A self-managed system or an autonomic system acts in the same way as a robot. A robot's Sense-Plan-Act (SPA) behavior corresponds exactly to a self-managed system (Kramer and Magee, 2007). This is why an autonomic system must achieve the same properties as for SPA architectures that can be expressed by a three-layered reference model approach described in (Kramer and Magee, 2007). This model does in general also refer to the research of artificial intelligence and mobile robots.

The reference model (**Table 1**) leads to the next section which introduces an autonomic-like system.

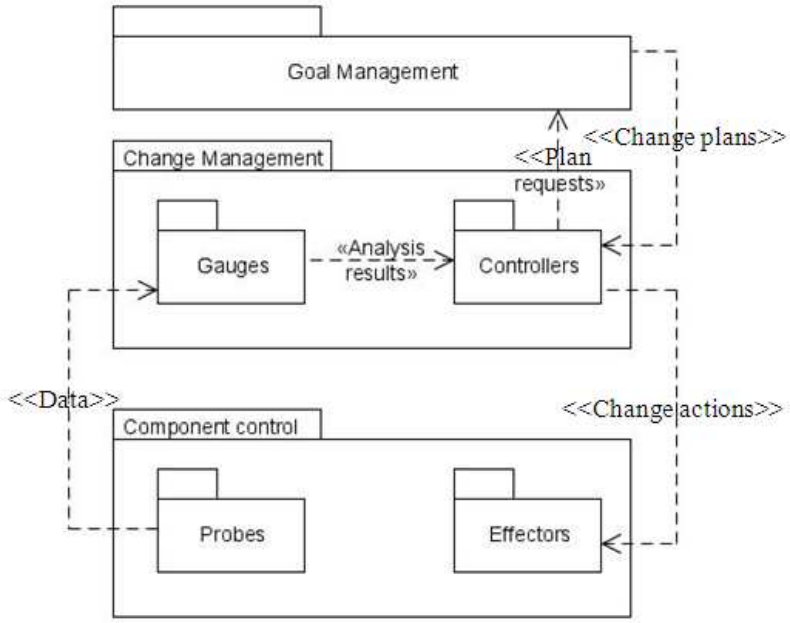

Fig. 2. The infrastructure placed in the three-layered architecture

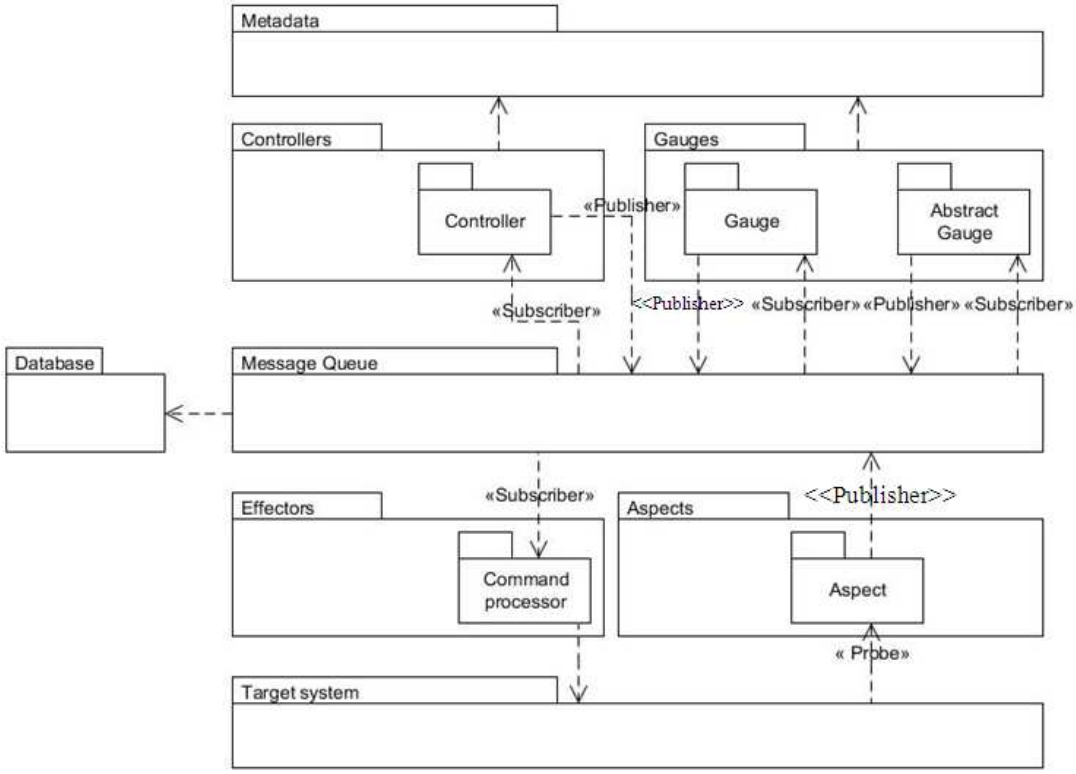

Fig. 3. Package structure

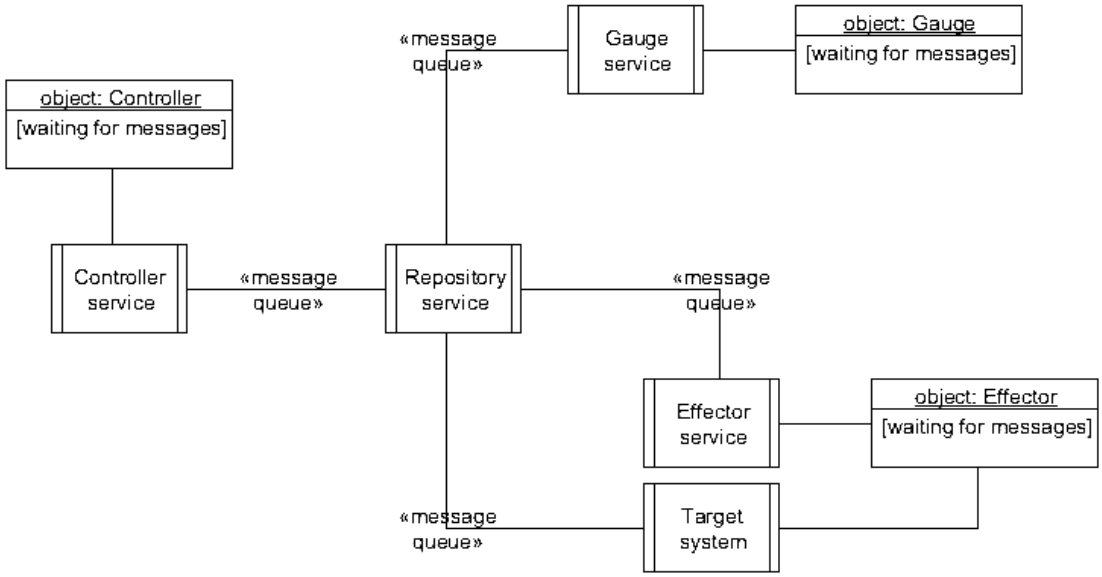

Fig. 4. Components and connectors

Table 2. Architectural assumptions (Bass *et al*., 2003)

Term	Definition
Requires assumptions	Detail services and resources that a component must have in order to correctly function
Provides assumptions	Describes the services a component provides to its users or clients

The components within the system will be identified respectively in correspondence to the respective layers in the reference model.

1.2. Monitoring Distributed Systems

Kaiser *et al*. (2003) an external infrastructure to monitor distributed legacy systems, named Kinesthetics Extreme, is made with the Java language. Although this work is centered on C++ language tools, its architectural overview and implementation is still relevant to this subject.

The approach is generally centered on two terms that are related to data collection and categorization, but only the architecture is interesting in relation to this work. The individual components (probes that collect, gauges that categorize and controllers that act) are loosely coupled by making them event-based. This is done by a standardized event middleware that uses the publisher/subscriber mechanism. All of these components can be placed respectively in the three-layered reference model as depicted in **Fig. 2**.

Furthermore, during the development of this infrastructure, learning techniques to build rules in a more autonomic fashion have also been investigated to address the upper-most layer.

1.3. Architectural Construction

The autonomic architectures that just have been introduced are used to form the architecture of this prototype with a functionality primarily centered on two use cases: The first use case includes the installation or log-enabling of a Windows Service. The second use case deals with the daily usage of the consolidated log.

The architecture in **Fig. 3** is generally like the Kinesthetics Extreme approach (Kaiser *et al*., 2003) with respect to the three-layered architecture. The reason is to achieve the same set of properties that makes it possible to detach components if not all of them are required or up scaling by attaching several components of the same type to increase performance.

The figure above (**Fig. 4**) illustrates how the components interact with each other through the message queue that holds messages in transit. It is generally implemented as a façade by a high level interface (Gamma *et al*., 1994), which is acting as a publisher/subscriber mechanism (Buschmann *et al*., 2007) through which the data is send to the components.

Commonly for third-party software is that it often have to match specific architectural assumptions (**Table 2**). The third-party software has to run under

certain requires assumptions and the provided assumptions do not always fit into the particular needs.

Architectural assumptions that do not properly fit into the architecture might often lead to architectural mismatches i.e., by introducing a component with non-scalable features as one of its provided assumptions into a scalable architecture.

Message queue software from third-party vendors exists, but these have been avoided. The reason for this is to avoid such architectural mismatches.

1.4. Collecting Low Level Data

Constructing an (abstract) architecture to make it fit into reference models and other architectural requirements is one thing; another thing is how to collect the data that need to be used with the architecture. The research of this subject includes in general techniques such as: collection of data, control theory, queuing models, heuristic search techniques and machine learning (Menasce and Kephart, 2007). However the following is centered on techniques used for collection of data:

- Data mining, but in order to do so the knowledge of all logs and their locations must be present along with parsers for each format
- Exception handling logs data a proprietary format to the applications own logs instead of a common log used for all applications which requires a parser for each format
- Dynamic probes are able to collect data in one format and broadcast the data to one common log, but it requires that the probes can be injected into the application

In general, data for error handling can be collected in many ways as outlined above, but the most appropriate one that is independent of various log files is the use of dynamic probes and data consolidation even the patterns for high level mapping must be defined either manually or automatically in some way.

Dynamic probes are primarily centered on collection of method calls that can be used to identify the states of the target system through pattern matching. The technique introduces data extraction by probes through code injection, which can be grouped into the following categories.

- Compile-time or static weaving
- Runtime or dynamic weaving

In general weaving is equivalent to code injection. The difference on compile-time weaving and runtime weaving relies on whether it is being processed on the source code of a non-running system or whether it is being injected into the binaries of a running system. A summary outlining the differences can be found below in **Table 3**.

Runtime weaving might be the only alternative hence it is not (always) possible to recompile and restart a mission critical system. However the target system can be statically prepared for the runtime techniques by making it aware of the aspects, which is why it is called prepared dynamic weaving.

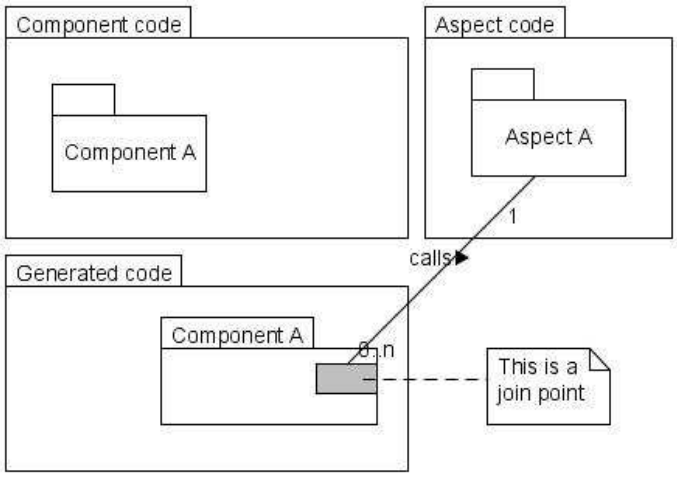

Fig. 5. The weaving aspect

Table 3. Summary of static- and dynamic weaving

Static weaving	Dynamic weaving
Requires declarations of elements to be visible to the source of the target system	Requires an expensive dynamic weaving structure
Source code needs to be available	Recompilation and restart of systems are not needed
Recompilation is required	

Table 4. Vocabulary defined by Aspect C++

Term	Definition
Join point	A join point refers to a point in the source code from which an aspect is accessed.
Point cut	A point cut is a set of join points described by a point cut expression
Advice	An advice declaration can be used to specify code that should run when the join points specified by a point cut expression are reached

Table 5. Vocabulary defined by DAC++

Term	Definition	Similar to
Minimal hook	A minimal hook represents a piece of code that allows our system to weave in an aspect at the hook location if needed	A join point

1.5. Aspect Weaving

Aspect weaving is a mix of two techniques; weaving and Aspect Oriented Programming (AOP). As its name implies, the use of aspect weaving introduces AOP as a single dimension of functional decomposition is insufficient to emphasis all aspects of a program in a modular way. AOP tries to solve this by encapsulating such crosscutting concerns in modular units.

In **Fig. 5**, it depicts the weaving aspect. The generated code is based on the original component code that has been merged with join points. These join points call the aspect code. As mentioned, this introduces a join point that leads to the definitions in **Table 4**, which is defined a compile-time weaving tool called Aspect C++ (Sincyk *et al.*, 2002). A few languages or language extensions support runtime weaving for the C++ language. Some languages support only dynamic weaving if the source has been prepared for it while others are able to make dynamic weaving without preparation, but they might instead have specific requirements to the compiled binaries.

1.6. Prepared Dynamic Aspect Weaving

The Dynamic Aspect C++ (DAC++) language (Almajali and Elrad, 2004), which is an extension to the C++ language, has been developed as part of a PhD program and the approach introduces the definitions outlined in **Table 5**.

The architecture of DAC++ uses two components: a preprocessor and an AOP engine. The preprocessor generates the Meta object data needed at runtime.

The prepared and compiled binary can be run with metadata containing information about program classes and methods. This information is used by the dynamic aspects called through minimal hooks and a designated AOP engine when weaving at runtime.

An evaluation with 100,000,000 iterations made on Linux G++ and DAC++ gave the same performance in benchmark tests. The difference in performance (**Table 6**) started appearing when we tested the performance of method calls as depicted in the taxonomy above.

1.7. Constructing Dynamic Aspects

The aspects are formed as modular units hence their primary responsibility is to collect data from all parts of the target system, which makes the responsibility a crosscutting concern. The aspect in **Table 7** traces all on-entry function calls. The amount of traces that are published can be reduced by either a filter or another and more strict point cut expression. However, such filtering mechanisms have been omitted. The validity of the aspect is described by a point cut expression and the weaving type which must be defined in the target system. The advice method (**Table 8**) is acting as a triggering event on the aspect.

The aspects are setup in a small piece of code (**Table 9**) compiled into the target system. It describes under which conditions the aspect should be working.

After the minimal hooks have been injected into the target system, it results in a code snippet such as the one in **Table 10**.

Table 6. Method Call Performance Evaluation

Benchmark	Hard coded aspect	DAC++
Method Call 1 (small)	32 sec	42 sec 31%
Method Call 2 (average)	36 sec	44 sec 22%

Table 7. Aspect, header information

Aspect	class TraceAllAspect : public Aspect {
	public:
	void advice() ;
	};

Point cut expression	Collects entries of all class method calls *

Table 8. Aspect, implementation details

Advice	void advice() {
	std::stringstream ss;
	// 1. use inherited methods to retrieve class-
	// and method names
	ss << /* xml construct omitted */ << "\n";
	ofstream File("c:\\message.xml");
	File.write (ss.str().c_str(),strlen(ss.str().c_str()));
	File.close();
	// 2. call AspectExe to publish on the message queue
	const char* szAspectExec =
	"C:\\...\\AspectsExecutable.exe c:\\message.xml";
	try { system(szAspectExec); } catch (...) {
	}
	}

Table 9. Enabling the aspect in the target system

Setup	DAC_INIT();
	// 1. defining point cut expression
	MethodPC meth;
	ClassPC classpc;
	meth.setvalue("*");
	classpc.setvalue("*");
	PCD pc;
	pc.setmethodPC(meth);
	pc.setclassPC(classpc);
	// 2. trace on entrance or exit of method calls
	WeaveSpecs ws;
	ws.setweavetype(Before_T);
	// 3. apply it to the respective aspect
	TraceAllAspect b;
	b.setweavespecs(ws);
	b.setpcd(pc);

1.8. Generating High Level Events

The techniques described collect data such as method calls from a running application. The collected data is very low level. This makes it difficult for usage on high level decision making in the daily work of the systems management staff.

Table 10. After the injection of join points for the run() method

run()	public : virtual void run ( ) { _dac_arrayvoidptr
	_dac_array_act_rec;
	_dac_arrayvoidptrnodeptr _dac_ptr2 ,_dac_ptr3;
	_dac_ptr3=NULL;
	if (MethodMOP[15].before)
	{ Aspectnodeptr _dac_ptr1;
	_dac_ptr1 = MethodMOP[15].beforeHead; while
	(_dac_ptr1!=NULL)
	{ _dac_execute_before(15,_dac_ptr1,_dac_ptr3);
	_dac_ptr1=_dac_ptr1->next; }
	} _org_run ();
	if (MethodMOP[15].after)
	{ Aspectnodeptr _dac_ptr1;
	_dac_ptr1 = MethodMOP[15].afterHead; while
	(_dac_ptr1!=NULL)
	{ _dac_execute_after(15,_dac_ptr1,_dac_ptr3);
	_dac_ptr1=_dac_ptr1->next; }
	}
	}

Table 11. Bridging low level data to high level events

Primitive category	Pattern	Abstract category
Executable	Sequence no.	Event name
Process id	Sequence size	Description
Thread id		
Class name		
Method name		

Table 12. The time (in seconds) it takes to process 226 traces, when the number of gauges is increased by a certain factor

Factor	Events	Actions	Total
1	57	12	69
2	31	6	37
3	23	3	26
4	19	3	22

To solve this it is necessary to map the low level data into a high level model. This is a time consuming process, it can either be done manually or in an autonomic-like way. However the patterns have been constructed manually in this prototype, but it is recommended that the respective providers of Windows services make this mapping in advance either in a proprietary format or in a standardized format that might include recovery procedures for the error states.

All method calls belong to sequences (or mapping traces) which define patterns. The patterns identify events that signal the states of the system such as "lost connectivity" among others.

The research outlined in (Walker *et al*., 2000) is primarily a target for construction of architectural views by

constructing a set of patterns matching events equal to the states of the target system. The part of the research that will be included is primarily the part about mapping traces.

The mapping trace that focus on events, such as class method entries and exits, are based on an encoding scheme which defines certain encoding events. These events are based on the determination of the patterns, where the patterns are based on sets or sequences of method calls. The mapping to the high level states bridges a set of primitive categories (such as class identifiers) to an abstract category through a partial, ordered specification of matching criteria defined by patterns. In other words, primitive categories are grouped into sequences that define a pattern. The patterns are matching abstract categories in the form of high level events (**Table 11**).

1.9. Performance and Adaptability

This section primarily focuses on two aspects: performance and adaptability due to large heterogeneous environments. Performance is evaluated by measuring the throughput based on two different Windows services, while the adaptability is evaluated by proving the prototype on a third-party Windows service with a different design.

The Windows Service has generated 226 traces within 59 seconds. The test of the other components is based on these 226 traces, where it is measured how long it takes to process these traces. The result of the test can be found in **Table 12**.

The adaptability issue has been proven by log-enabling a small third-party telnet service. It can be downloaded from codeproject.com (More, 1999) and it is called NT Telnet server and client. It is a tiny Windows Service created in the C++ language.

2. CONCLUSION

A prototype was made to clarify whether it was possible to generate a consolidated log of high level events based on low level traces from various Windows services. The prototype was based on a three-layered autonomic reference model to allow more automated features in the prototype.

In order to clarify such consolidated log, the correctness has been evaluated. It has been evaluated from two points of view: First, the consolidated log and the proprietary logs must signal the same states. Second, the same type of errors (expressed as high level events) must be homogeneous regardless of which Windows services that have been generating them.

However the biggest issue is the generation of patterns. It is time consuming and requires special knowledge of the systems. Second to this, the preparation of the Windows services requires knowledge about the source code, compilers and compiler environment.

3. ACKNOWLEDGEMENT

Thanks to everyone who helped with this article. Especially thanks to Localization Manager Vibeke Batchford at Trapeze Group Europe A/S for proofreading this article. She holds a MA degree in English. Finally, thanks to both Trapeze Group Europe A/S and Princess Sumaya University for the economical contributions.

4. REFERENCES

Almajali, S. and T. Elrad, 2004. A dynamic aspect oriented C++ using mop with minimal hook weaving approach. Proceedings of the Dynamic Aspect Workshop, Lancaster, (DAWL' 04), England.

Bass, L., P. Clements and R. Kazman, 2003. Software Architecture in Practice. 2nd Edn., Addison Wesley, Boston, ISBN-10: 0321154959, pp: 560.

Buschmann, F., K. Henney and D.C. Schmidt, 2007. Pattern Oriented Software Architecture. 1st Edn., Wiley, Chichester, England, ISBN-10: 0470512571, pp: 490.

Gamma, E., R. Helm and R. Johnson, 1994. Design Patterns: Elements of Reusable Object-Oriented Software. 1st Edn., Addison-Wesley, Reading, Mass, ISBN-10: 0201633612, pp: 395.

IBM, 2005. An Architectural Blueprint for Autonomic Computing. 3rd Edn., Autonomic Computing, pp: 34.

Kaiser, G., J. Parekh, P. Gross and G. Valetto, 2003. Kinesthetics extreme: An external infrastructure for monitoring distributed legacy systems. Proceedings of the Autonomic Computing Workshop, Jun. 25, IEEE Xplore Press, pp: 22-30. DOI: 10.1109/ACW.2003.1210200

Kramer, J. and J. Magee, 2007. Self-managed systems: An architectural challenge. Future Software Eng. DOI: 10.1109/FOSE.2007.19

Menasce, D.A. and J.O. Kephart, 2007. Autonomic computing. IBM Res. IEEE Comput. Soc.

More, R., 1999. NT telnet server and client.

Sincyk, O., A. Gal and W. Schroder-Preikschat, 2002. AspectC++: An aspect-oriented extension to the C++ programming language. Proceedings of the 14th International Conference on Tools Pacific: Objects for Internet, Mobile and Embedded Applications, (CRPIT' 02), ACM Press, Australia, pp: 53-60.

Walker, R.J., G.C. Murphy, J. Steinbok and M.P. Robillard, 2000. Efficient mapping of software system traces to Architectural views. Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research, (CASCON' 00), ACM Press, pp: 12-12.